

Institute of Parallel and Distributed Systems

University of Stuttgart
Universitätsstraße 38
D-70569 Stuttgart

Bachelorarbeit Nr. 281

Bootstrapping ontology-based data access specifications from relational databases

Philipp Martis

Course of Study: Informatik

Examiner: PD Dr. Holger Schwarz

Supervisor: M. Sc. Leif Harald Karlsen

Commenced: 23rd of November 2015

Completed: 24th of May 2016

CR-Classification: D.0, H.2.8

Abstract

Ontology-based data access (OBDA) is a useful approach to gain access to existing structured data in an easy and intuitive manner, abstracting from the representation of the data and thus releasing from the burden of having to deal with technical details of the data structuring [SGH⁺15].

By using the approach of ontology bootstrapping, an ontology reflecting the structure of the data can automatically be created by examining the data source [SGH⁺15] [MNS03]. This allows to benefit from ontology-based data access concepts without having to forgo the advantages regarding performance and data consistency which is offered by relational database systems. Furthermore, establishing a system providing a so-called “virtual RDF view”, data from relational sources can be accessed and even updated by querying or modifying the ontology while remaining in the database [EK12].

However, the creation of an ontology from another structured data source is a complex task that involves several tools, formats and languages. Also taking changing input data into account, the process of ontology bootstrapping becomes a significant engineering task in its own right [SGH⁺15]. The approach of introducing OBDA specifications proposed by Skjæveland et al. address this issue by concentrating all relevant information on the bootstrapping process in one place, allowing for systems which drive the entire tool chain based on this information and thus rendering the use of ad-hoc scripts unnecessary [SGH⁺15]. A concrete format for serializing such an OBDA specification for storage or exchange, however, is not provided.

The introduction of such a format is part of this thesis. After the task it accomplishes, the format was named OBDA SPECIFICATION LANGUAGE (OSL). Furthermore, this thesis describes the design and implementation of DB2OSL, a software which is able to, in turn, automatically bootstrap an OBDA specification from a relational database schema. Finally, it gives an explanatory description of both the process of ontology bootstrapping and the process of an OBDA specification and proposes a refined scheme for generating IRIs for the entities of the resulting ontology.

Kurzfassung

Ontologiebasierter Datenzugriff (OBDA von engl. ‘Öntology-based data access’) ist ein nützlicher Ansatz um auf einfache und intuitive Weise Zugriff auf gegebene strukturierte Daten zu erhalten, der von der konkreten Representation der Daten abstrahiert und deshalb die Bürde vom Benutzen nimmt, sich mit technischen Details der Datenunterteilung beschäftigen zu müssen [SGH⁺15].

Durch den Ansatz der Ontologieableitung kann durch eine Untersuchung der Quelldaten automatisch eine Ontologie erstellt werden, die die Struktur der Daten widerspiegelt [SGH⁺15] [MNS03]. Das erlaubt es, von den Vorteilen des ontologiebasierten Datenzugriffs zu profitieren, während die Vorteile von relationalen Datenbanksystemen hinsichtlich der Performanz und Datenkonsistenz nicht aufgegeben müssen. Des Weiteren kann durch den Aufbau eines Systems mit einer so genannten ‘virtuellen RDF-Ansicht’ auf Daten einer relationalen Datenbank zugegriffen und diese sogar verändert werden indem Anfragen oder Modifikationen an die bzw. der Ontologie durchgeführt werden, während die Daten in der Datenbank bleiben [EK12].

Jedoch ist die Ableitung einer Ontologie aus einer anderen Form strukturierter Daten eine komplexe Aufgabe, die verschiedene Werkzeuge, Formate und Sprachen involviert. Die zusätzlichen Anforderung sich ändernder Quelldaten macht den Prozess der Ontologieableitung vollends zu einer eigenen technischen Herausforderung [SGH⁺15]. Der von Skjæveland et al. vorgeschlagene Ansatz, OBDA-Spezifikationen einzuführen, begegnet dieser Problematik, indem die für den Ableitungsprozess relevanten Informationen an einer Stelle gesammelt werden und so Systeme ermöglichen, die anhand dieser Informationen den gesamte Werkzeugverband steuern, wodurch zusammengestückelte Systeme redundant werden [SGH⁺15]. Ein genaues Format um diese OBDA-Spezifikationen zu speichern und auszutauschen wird jedoch nicht angegeben.

Die Einführung eines solchen Formats ist Teil dieser Arbeit. Nach ihrer Aufgabe wurde das Format ‘OSL’ genannt (deutsch: ‘ÖBDA-Spezifikationssprache’). Außerdem beschreibt diese Arbeit das Design und die Implementierung von DB2OSL, einer Software, die in der Lage ist, wiederum die Ableitung einer OBDA-Spezifikation aus einem relationalen Datenbankschema automatisch zu bewerkstelligen. Zudem enthält sie eine erklärende Beschreibung sowohl der Ontologieableitung als auch der Ableitung einer OBDA-Spezifikation und schlägt ein verfeinertes Schema zur Generierung von IRIs für die Elemente der Zielontologie vor.

Contents

Abstract	iv
Kurzfassung	v
Contents	vii
List of figures	viii
List of tables	ix
1. Introduction	1
1.1. Motivation	1
1.2. Approach	1
1.3. Requirements and goals	2
2. Background and related work	4
2.1. Background	4
2.1.1. Basic concepts	4
2.1.2. Ontology-based data access (OBDA)	4
2.1.3. OBDA specifications	6
2.1.4. The OPTIQUE project	7
2.2. Related work	7
2.2.1. Ontologies and the semantic web – publications	7
2.2.2. OBDA specifications – publications	7
2.2.3. General ontology bootstrapping – publications	8
2.2.4. The OPTIQUE project – publications	8
2.2.5. Alternative approaches – publications	8
3. On bootstrapping and IRI generation	9
3.1. Ontology bootstrapping using direct mapping	9
3.1.1. Overview on the direct graph	9
3.1.2. Data representation in direct mapping	10
3.1.3. IRI generation in direct mapping	10
3.1.4. Conclusion on direct mapping	11
3.2. Ontology bootstrapping using OBDA specifications	11
3.2.1. Structure of OBDA specifications	12
3.2.2. Using OBDA specifications	13
3.2.3. Bootstrapping OBDA specifications	18
3.2.4. Conclusion on OBDA specifications	20

3.3.	Generating unique IRIs for OBDA specification map fields	21
3.3.1.	Requirements for the IRI scheme	22
3.3.2.	Avoiding name clashes in the IRI scheme	22
3.3.3.	The proposed IRI generation scheme	24
3.3.4.	Proof of correctness of the proposed IRI scheme	26
3.3.5.	Conclusion on the IRI generation scheme	26
4.	The OBDA Specification Language (OSL)	27
4.1.	Requirements on the OBDA SPECIFICATION LANGUAGE (OSL)	27
4.2.	Design of the OBDA SPECIFICATION LANGUAGE (OSL)	27
4.2.1.	IRIs in the OBDA SPECIFICATION LANGUAGE (OSL)	29
4.3.	Specification of the OBDA SPECIFICATION LANGUAGE (OSL)	29
4.4.	An example OSL specification	32
4.4.1.	The example database schema	32
4.4.2.	The resulting example OSL specification	32
4.5.	Conclusion on the OBDA SPECIFICATION LANGUAGE (OSL)	36
5.	The db2osl software	38
5.1.	Requirements on DB2OSL	38
5.2.	Functionality	38
5.3.	Command line interface	40
5.3.1.	Command-line arguments	40
5.3.2.	Multiple bootstrapping operations or multiple servers	42
5.4.	Usage and integration	43
5.4.1.	Basic usage	43
5.4.2.	Handling OSL specifications	43
5.4.3.	Integrating DB2OSL into systems	44
5.5.	Architecture of DB2OSL	45
5.5.1.	Overall proceeding of DB2OSL	45
5.5.2.	Libraries used in DB2OSL	46
5.5.3.	Coarse structuring of DB2OSL	47
5.5.4.	Fine structuring of DB2OSL	52
5.6.	Conclusion on DB2OSL	58
6.	Implementation of db2osl	60
6.1.	Tools employed to create DB2OSL	60
6.2.	Code robustness	61
6.2.1.	Enforcing error detection	62
6.2.2.	Const correctness	62
6.3.	Code readability	63
6.3.1.	Comments	64
6.3.2.	Expressive identifier names	65
6.3.3.	Intuitive use of classes	65
6.3.4.	Avoidance of unnecessary interfaces	66
6.3.5.	Intuitive use of packages	66
6.3.6.	Intuitive control flow	67
6.3.7.	Usage of well-known structures	68

7. Summary and future work	69
7.1. Summary	69
7.2. Future work	70
Appendix A. Details on the db2osl implementation	72
A.1. Package contents (DB2OSL)	72
Appendix B. Details on the OBDA Specification Language (OSL)	73
B.1. The OSL header	73
Bibliography	84

List of Figures

1.1.	Illustration of the overall bootstrapping process	2
2.1.	The two basic OBDA system architectures	6
a.	OBDA system architecture with materialized RDF view	6
b.	OBDA system architecture with a virtual RDF view	6
3.1.	Constitution of the <i>direct graph</i>	10
3.2.	Constitution of an OBDA specification	13
5.1.	Illustration of the overall proceeding of DB2OSL	45
5.2.	Exemplification of an extended DB2OSL	46
5.3.	Package dependencies in DB2OSL	50
5.4.	Package dependencies in earlier versions of DB2OSL	51
5.5.	Database class hierarchies in DB2OSL	53
a.	ColumnSet class hierarchy in DB2OSL	53
b.	ColumnSet class hierarchy in DB2OSL – simplified	53
c.	Column class hierarchy in DB2OSL	53
5.6.	OBDA specification class hierarchies in DB2OSL	54
a.	URIBuilder class hierarchy in DB2OSL	54
b.	OBDAMap class hierarchy in DB2OSL	54
5.7.	Logging and output class hierarchies in DB2OSL	55
a.	SpecPrinter class hierarchy in DB2OSL	55
b.	StreamHandler class hierarchy in DB2OSL	55
5.8.	Job class hierarchy in DB2OSL	56
5.9.	Miscellaneous class hierarchies in DB2OSL	57
a.	Iterable class hierarchy in DB2OSL	57
b.	ReadOnlyIterator class hierarchy in DB2OSL	57
c.	Iterator class hierarchy in DB2OSL	57
d.	Exception class hierarchy in DB2OSL	57
e.	RuntimeException class hierarchy in DB2OSL	57

List of Tables

3.1.	Assignment of values to fields of OBDA specification maps	19
3.2.	Proposed IRIs to be used in OBDA specification map fields	25
4.1.	OWL individual IRIs in OSL	30
4.2.	OWL class membership of map representations in OSL	30
4.3.	OWL property IRIs in OSL	31
5.1.	Command-line arguments in DB2OSL – descriptions	41
5.2.	Command-line arguments in DB2OSL – default values	42
5.3.	Descriptions of the packages in DB2OSL	48
5.4.	Standalone classes in DB2OSL	58
A.1.	Class attachment to packages in DB2OSL	72

1. Introduction

1.1. Motivation

As estimated in 2007 [HPZC07], publicly available databases contained up to 500 times more data than the static web, and roughly 70 % of all websites were backed by relational databases at that time. As hardware has become cheaper yet more powerful, open source tools have become more and more widespread and the web has become more and more dynamic and interactive, it's likely that these numbers have even increased since then. This makes the publication of available data in a structured, machine-processable form and its retrieval with eligible software an interesting topic. The most important formalism to represent structured data without the need of a fixed (database) schema is ontologies, and thus this approach is known under the term “Ontology-based data access” (“OBDA”).

The vision of a machine-processable web emerged as early as 1989 [BL89] and was entitled with the term “semantic web” by Tim Berners-Lee in 1999 [BLF99]. Definitely, the automatic translation of relational databases to RDF [W3C14b] or similar representations of structured information is an integral part of the success of the semantic web [HPZC07]. This automatic translation process is commonly called “bootstrapping”.

Today, the pure bootstrapping process is a relatively well understood topic, ranging from the rather simple direct mapping approach [W3CR12a] to the approach by Maedche et al. using machine learning techniques [MNS03]. On the other hand, the handling of the complexity introduced by these approaches and the use of sophisticated tools to perform various related tasks meanwhile has become a significant challenge in its own right [SGH⁺15]. Besides the parametrization of the tools in use, this includes the management of the several kinds of artifacts accruing during the process, possibly needed in different versions and formats for the use of different tools and output formats, while also taking changing input data into account [SGH⁺15]. Skjæveland and others therefore suggested an approach using a declarative description of the data to be mapped, concentrating in one place all the information needed to coordinate the bootstrapping process and to drive the entire tool chain [SGH⁺15].

1.2. Approach

This thesis describes the development of a specification language to serialize the declarative specification of the bootstrapping process (see Section 1.1 – Motivation) and of a software to, in turn, bootstrap it from a relational database schema. In light of the tasks they accomplish, the specification language was called “OBDA Specification Language” (“OSL”) and the software bootstrapping the specification was called “db2osl”. They are described in Chapter 4 – The OBDA SPECIFICATION LANGUAGE (OSL), Chapter 5 – The DB2OSL software

and Chapter 6 – [Implementation of DB2OSL](#).

Furthermore, this thesis suggests a scheme for generating the IRIs that occur in OBDA specification, identifying their parts [SGH⁺15]. Currently, this issue is only exemplified and there is room for improvement in the sense that a simple and straight-forward approach can be used to generate IRIs for all constituents of OBDA specifications without introducing name clashes in corner cases. This approach is explained in Section 3.3 – [Generating unique IRIs for OBDA specification map fields](#).

Using a declarative specification makes the entire bootstrapping process a two-step-procedure, as illustrated in Figure 1.1: First, the OBDA specification is derived from the database schema using DB2OSL. It specifies the actual bootstrapping process in a very general way, so it only has to be recreated when the database schema changes. The second step is to use the OBDA specification to coordinate and drive the actual bootstrapping process. The development of a software that uses the OBDA specification to perform this second step currently is subject to ongoing work. It will be able to be parameterized accordingly to support different output formats, tools, tool versions and application ranges.

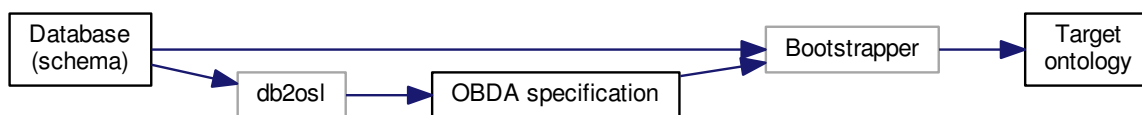


Figure 1.1.: Illustration of the overall bootstrapping process using a declarative OBDA specification

1.3. Requirements and goals

The final system shall be able to cleanly fit into existing bootstrapping systems while being easy to use, relieving its users of the burden of dealing with OSL specifications manually instead of adding even more complexity to the process. To achieve these goals, use of existing tools, languages and conventions was made wherever possible. For example, the OBDA SPECIFICATION LANGUAGE was defined to be a subset of OWL. This facilitates meeting the objective of a powerful, easy-to-use, flawless and well-documented language that can be extended and handled by existing tools.

To fit into the environment used in the OPTIQUE project [CGH⁺13] of which it is ultimately part, JAVA was used for the bootstrapping software. Specific care was taken to design it to be modular and flexible, making it usable not only as a whole but also as a collection of independent components, possibly serving as the basis for a program library in the future. To achieve this objective and to make the software more easily understandable and extendable, careful and thorough documentation was carried forth.

As the software will be maintained by diverse parties after its development and will likely be subject to changes, general code quality was also an issue to consider. Following good object-oriented software development practices [Str00], real world artifacts, like database schemata, database tables, columns, keys, and OBDA specifications were modeled as software objects and provided with a carefully chosen set of operations to manipulate them and make them collaborate. This approach and other actions aiming at yielding clean code are described

more thoroughly in Chapter 6 – [Implementation of DB2OSL](#), while the resulting structure of the software is discussed in Section 5.5 – [Architecture of DB2OSL](#).

2. Background and related work

2.1. Background

2.1.1. Basic concepts

The basic concepts underlying ontology-based data access, since being too extensive and probably known to most readers, are not described in this thesis.

For an introduction to these concepts, however, refer to the literature referenced in Section 2.2.1 – [Ontologies and the semantic web – publications](#).

2.1.2. Ontology-based data access (OBDA)

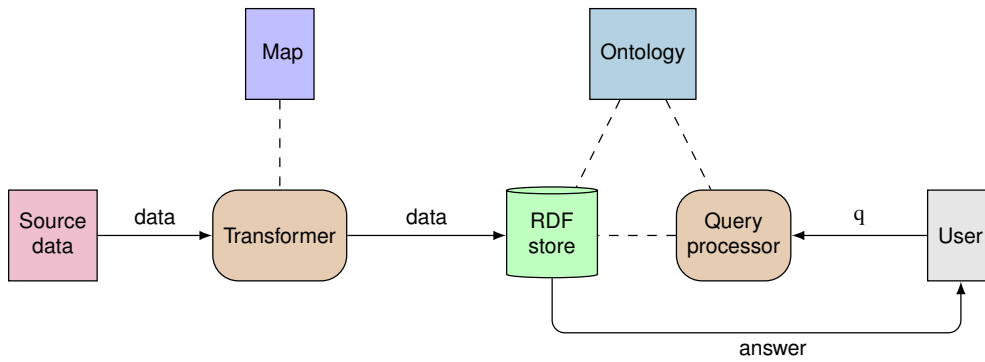
Storing data in relational databases is a very common proceeding, since the notion of a relational database is comprehensible and widely known, and the required software is widely available both commercially and as open-source software. Thus, it is easy for a domain expert to set up and populate a database. Furthermore, relational databases provide significant advantages concerning performance, data consistency and integrity, integration abilities, support and general prominence. These topics definitely played a major role in the success and the extensive research that databases discovered, up to the degree that these are the main fields the strengths of databases are seen in. Many – if not all – of these strengths trace back to the relatively fixed and rigid structure that databases embody: a well-defined database schema imposing strong and clear cut constraints on the contained data.

However, this principle also induces notable disadvantages. The database schema, although theoretically changeable, constitutes a significant burden on the representation of data of dynamic environments, incomplete data or changing requirements, especially when dealing with large amounts of data. The resulting representation of data in unintuitive, suboptimal schemata makes the use of prolonged and complex query constructs inevitable, which lets more elaborate queries quickly become unmanageable for non-experts and time-consuming and error-prone, even for experts. Ontologies, on the other hand, are significantly more flexible regarding incomplete data or changing requirements or environments and allow for considerably more intuitive and abstract query systems, while still being a quite comprehensible formalism (see Section 2.2.1 for publications describing ontologies and the semantic web). Besides, ontologies provide support for different data records referencing the same entity, while databases do not [SGH⁺15]. Ontologies also provide support for the deduction of implicit information [SGH⁺15], which with common database systems, is – at the very least – not possible out of the box and in an easy manner, if it is possible at all. Often, however, relational databases are preferred for their advantages (although the availability of cheaper yet more powerful

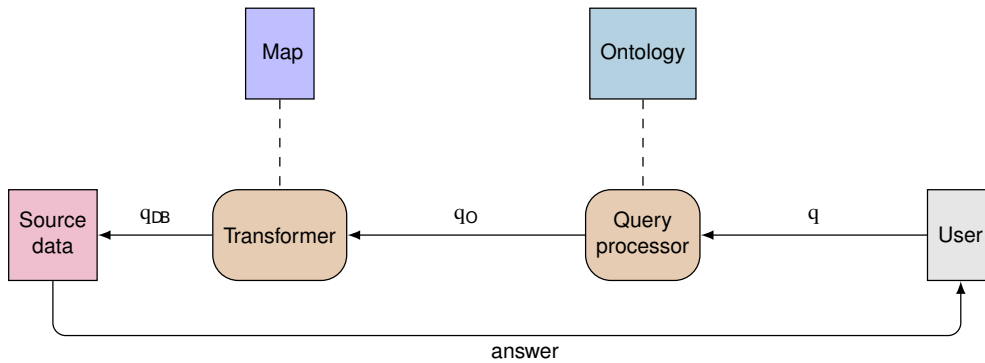
hardware in some cases offsets these), and sometimes they are even preferred due to lack of knowledge. Besides, in some cases, the migration to ontology-based systems, even if beneficial, is too costly to be seriously considered.

Ontology-based data access often provides a solution to this collision of interests: By adding an ontology-based front-end processing the queries that is sensibly mapped to the data representation, the querying facilities of ontology-based systems are made available. In such a system, changes in the data representation most often can be carried out without breaking existing queries; only the mapping has to be changed. Furthermore, changes in mappings is a one-time effort. Only when it introduces changes in the way in which it presents existing structures to the user, existing queries have to be modified. The creation of these mappings in turn can happen computer-aided or, in simple cases or to a certain degree of completeness and accuracy, the mappings can be completely bootstrapped.

Moreover, in cases where it is too costly or for other cases unfeasible to carry out a complete data duplication, the data can remain in the underlying database as is, and the query front-end merely acts as an interface transforming the query into a database query [SGH⁺15] by making use of a backward-chaining technique called *query rewriting* [SGH⁺15]. This approach is called *virtual OBDA* or *virtual RDF view* [SGH⁺15] and is illustrated in Figure 2.1b. The oppositional approach of duplicating all data is called *materialized OBDA* or *materialized RDF view* and is illustrated in Figure 2.1a. Virtual OBDA provides limited abilities compared to materialized OBDA in that it does not allow for decoupling from the source data by, for example, adding inferred information or applying elaborate transformations and does not support “fragments of OWL for which query rewriting is not a complete deduction method” [SGH⁺15]. However, the response time of systems is hard to predict solely from the architectural approach used, so if it is critical, several systems should be prototyped and evaluated upfront on what are expected to be typical queries [SGH⁺15].



(a) OBDA system architecture with materialized RDF view



(b) OBDA system architecture with a virtual RDF view

Figure 2.1.: The two basic OBDA system architectures: materialized and virtual RDF views (from [SGH⁺15])

Finally, it has to be mentioned, that ontology-based data access is not limited to databases. Although this is the most common scenario and the only one this thesis deals with, ontology-based data access also works with other sources of structured information, like ODS, XLS or CSV files, though some additional preparation might be necessary in these cases [SGH⁺15].

2.1.3. OBDA specifications

As mentioned in Section 1.1 – Motivation, the sole bootstrapping of RDF triples [W3C14b] or other forms of structured information from relational database schemata is a relatively well understood topic. This is outlined more comprehensively in Section 2.2 – Related work.

Nonetheless, bootstrapping remains an elaborate process involving complex tools to be invoked – possibly in different versions and configurations and processing different formats – and working on changing input data [SGH⁺15]. This is why Skjæveland and others proposed the introduction of OBDA specifications centralizing the task of driving these tools and to gather in one place all the information describing the desired mapping between the source database and the target ontology [SGH⁺15]. As described in Section 2.2 – Related work, their approach is the foundation of this thesis, which describes and specifies a format for storing and exchanging such OBDA specifications – the OBDA SPECIFICATION LANGUAGE (OSL) – and introduces

a tool that, in turn, automatically bootstraps OBDA Specifications from relational database schemata – the DB2OSL software.

The bootstrapping process using OBDA specifications and DB2OSL is illustrated in Figure 1.1 in Section 1.2 – Approach.

2.1.4. The OPTIQUE project

The problems addressed in Section 1.1 – Motivation constitute a big issue inter alia in the oil and gas industry: 30% to 70% of the working time of engineers is spent on collecting data or assessing its quality [Cro08]. This led to the origination of the OPTIQUE project which “advocates for a next generation of the well known Ontology-Based Data Access (OBDA) approach to address the data access problem [...] [aiming] at solutions that reduce the cost of data access dramatically” [KGJR+13]. Thus, the OPTIQUE project tries to obtain precisely the benefits a well-developed OBDA system can provide (explained in Section 2.1.2): an easy end-user access to data without knowing about its structuring while taking advantage of automatic translations [CGH+13]. In doing so, ascertained shortcomings of existing OBDA systems were addressed: *usability* (for example the need to use formal query languages), *costly prerequisites* (consider, for example, the disadvantages of materialized OBDA described in Section 2.1.2) and *efficiency* (which was perceived as being insufficiently addressed in previous approaches) [KGJR+13].

2.2. Related work

2.2.1. Ontologies and the semantic web – publications

Uschold et al. [UG+96] provide an introduction into ontologies and a description of the underlying concepts.

For an definitions and explanations of URIs and IRIs, see [DS04].

For a description and specification of RDF by the W3C RDF Working Group, see [W3C14b].

For a description and specification of RDFS by the W3C RDF Working Group, see [W3C14a].

For a description and specification of OWL, see [W3C12].

For a description and specification of R2RML Mapping Language by the W3C RDB2RDF Working Group, see [W3CR12b].

2.2.2. OBDA specifications – publications

A publication building the foundation of the work presented in this thesis is the summarizing and benchmarking work on OBDA specifications by Skjæveland et al. [SGH+15], the group that developed them in their present form.

2.2.3. General ontology bootstrapping – publications

Skjæveland, Lian and Horrocks [SLH13] provided an exemplifying description of the transformation of the *NPD FactPages*, an enormous collection of data related to oil drilling on the Norwegian continental shelf, provided by the Norwegian Petroleum Directorate (NPD).

Sequeda et al. [STCM11] provided an overview over different direct mapping approaches.

Sequeda, Arenas and Miranker [SAM12] [SAM11] describe the direct mapping of relational databases to RDF and OWL formally.

Stojanovic, Stojanovic and Volz [SSV02] published a formal description of the mapping of relational databases onto ontology-based structures, describing concepts preceding and/or supplementing OWL and using F-LOGIC [KL89] as target language.

2.2.4. The OPTIQUE project – publications

Calvanese et al. [CGH+13] presented the OPTIQUE project including its underlying OBDA system and showed limitations of current OBDA systems.

Kharlamov et al. [KGJR+13] described the first version of the OPTIQUE system, customized for use with the *NPD FactPages* of the Norwegian Petroleum Directorate (NPD).

Skjæveland and Lian [SL13] summarized the benefits of and the proceeding for converting the NPD FACTPAGES to linked data [BHBL09] and discuss associated terms like linked data, URIs, RDF and SPARQL.

2.2.5. Alternative approaches – publications

Barrasa, Corcho and Pérez [BRCGP04] proposed a declarative mapping language – R2O – which is capable of expressing a mapping between a relational database and ontologies represented in the OWL and RDF formats. This approach however aims at connecting existing databases and existing ontologies.

Maedche et al. [MNS03] proposed an approach for automatic bootstrapping which makes use of machine learning techniques.

More approaches are described in the survey by Sahoo et al. in [SHH+09].

3. On bootstrapping and IRI generation

3.1. Ontology bootstrapping using direct mapping

Direct mapping describes a way to map relational data to ontologies. As such, it is not an alternative to using OBDA specifications but rather a foundation to base the mapping approach on. The direct mapping approach aims at providing exactly such a foundation, offering extension points for refinement at several places [W3CR12a]. Furthermore, as the name suggests, it aims at being a simple, comprehensible and straight forward approach. As with all ontology bootstrapping strategies, the resulting ontology describes the different types of data and how they are related to each other, including individuals arising from the input data records (for an explanation of the basic ontology concepts, refer to Section 2.1.1).

Direct mapping is currently a W3C recommendation, which defines the production of an RDF graph – which is called the *direct graph* – from a relational schema [W3CR12a]. As a matter of fact, the principal definition of the direct graph, excluding definitions of rather trivial subcomponents, fits on one computer screen. The direct graph contains all data held in the source database, but it does not contain additional schema information like uniqueness of or non-null constraints on columns [W3CR12a].

It is important to understand ontology bootstrapping using OBDA specifications, which is described in Section 3.2, is not an alternative approach to ontology bootstrapping using direct mapping, but a means to base the bootstrapping process on a well-defined description, yielding something similar to a direct mapping when processing this description properly. Which approach is ultimately used in the bootstrapping process also heavily depends on the population of this description, which is what this thesis mainly deals with. In particular, a high degree of freedom is provided in the generation of URIs, for which this thesis proposes a generation scheme enhancing the scheme used by the direct mapping approach (see Section 3.3). As the bootstrapping software for OBDA specifications introduced in Chapter 5 also generally creates OBDA specifications in a simple and direct way, and furthermore uses an URI generation scheme closely resembling the one proposed by direct mapping, it also can be seen as based on direct mapping. Because direct mapping is such an important foundation for the work described in this thesis, it is described in this extra section.

3.1.1. Overview on the direct graph

The constitution of the direct graph is illustrated in Figure 3.1. Its basic components are, for each row, the *row type triple*, its *literal triples* and its *reference triples*. Here, the row type triple encodes which table the respective row belongs to, the literal triples encode the data in non-foreign-key columns, and the reference triples encode the data in foreign key columns. These triples are then, by degrees, united to the direct graph: the row triples of each row form

the row graph, the row graphs of each table form the table graph and all table graphs united constitute the final direct graph.

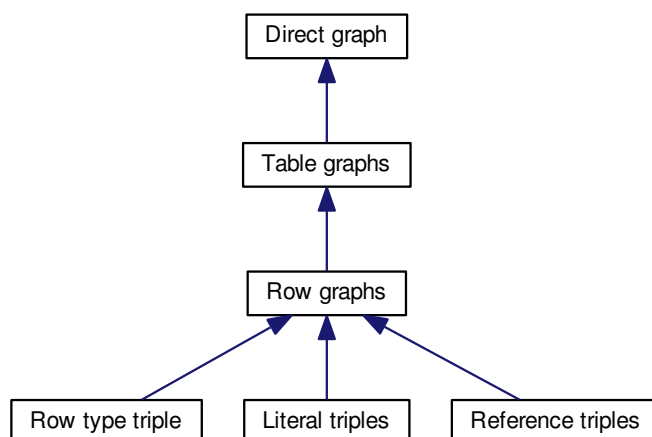


Figure 3.1.: Constitution of the *direct graph*. “ \rightarrow ” means “is part of”.

Carefully assigning IRIs to the RDF entities is an essential part of the approach, since otherwise, name clashes can occur. Indeed, there seems to be a corner case which was not considered. For details on IRI generation in direct mapping, see Section 3.1.3.

3.1.2. Data representation in direct mapping

Since the result of a direct mapping is an RDF graph, the means to represent data are limited to valid RDF vocabulary. This is no problem for IRIs and expressions that only involve IRIs (row type triples and reference triples).

To encode the non-foreign-key data content of the source database – literal triples – the R2RML mapping language is used, a language providing a mapping from the relational data model to the RDF data model [W3CR12b] (see Section 2.1.1 – Basic concepts for a short description). R2RML expressions thereby are themselves RDF statements. The data value contained in a direct mapping literal triple is defined to be the *R2RML natural RDF literal* representation of the value [W3CR12a], which, as the name suggests, is a single RDF literal [W3CR12b].

3.1.3. IRI generation in direct mapping

As all RDF triples generated by the direct mapping are simply united to constitute the final *direct graph* (see Section 3.1.1 – Overview on the direct graph for details), a proper IRI assigning is vital to the functioning of the approach. By design, IRIs for different kinds of entities have a different structure, which prevents name clashes on the one hand, but on the other hand induces that, in case of a clash, all entities with the conflicting IRI are of the same kind. This implies the existence of a high risk of producing ambiguous information and thus losing data.

The relatively simple way IRIs are assigned in direct mapping is described in the following [W3CR12a]:

- Table IRIs correspond to the table name.
- Literal property IRIs consist of the table name and the column name, separated by a hash character ('#').
- Reference property IRIs consist of the child table name, the string “#ref-” and the child table column names of the respective foreign key, separated by a semicolon (;).
- All contained names are included in their percent-encoded form.

The encoding of reference property IRIs can lead to name clashes in cases multiple foreign keys exist which contain exactly the same columns. This is allowed, for example, in SQL [VDL89]. To remove this flaw, the parent table name and the parent table column names of the respective foreign key must also be included in the IRI.

3.1.4. Conclusion on direct mapping

Direct mapping is a simple and straight-forward approach for ontology bootstrapping, which its main advantage. Furthermore, it is defined by the W3C in a public document, making use of other W3C definitions and specifications familiar to many users involved in the topic.

On the other hand, direct mapping suggests an insufficient URI generation scheme which introduces name clashes in the case that several foreign key restriction exists in a table which involve exactly the same columns (see Section 3.1.3). Because of this, it should not be used in its original form.

However, due to the conciseness of its definition, its status of being a W3C recommendation, and its directness, direct mapping reaches its objective of being a simple, widely available, and well-defined foundation for ontology mapping approaches.

3.2. Ontology bootstrapping using OBDA specifications

This section presents and explains the proceeding of bootstrapping ontologies for ontology-based data access with the use of OBDA specification. As mentioned in Section 3.1, bootstrapping using OBDA specifications is not an alternative or contrary approach to using direct mapping. Instead, OBDA specifications are used to specify and control the bootstrapping process, gathering momentous information at a central place. Thereby, they aim at greatly reducing the complexity introduced by different tools, formats and languages, rendering the use of ad-hoc scripts unnecessary and facilitating the integration of changing data and changing environments into the process. Additionally, their description by Skjæveland et al. [SGH⁺15] is accompanied by formal rules which describe a bootstrapping process similar to direct mapping, but extending it in many ways. For example, OBDA specifications and the formal rules to be applied to them, establish subclass-superclass relationships in the target ontology, which is not provided by the direct mapping approach.

As a matter of fact, one is not bound to a particular mapping approach when using OBDA

specifications, if the associated formal rules are not applied to them, and neither does a given OBDA specification prescribe which approach to use. In fact, they provide a high degree of freedom in the utilization of URIs: custom URIs or URI patterns can be generated and specified for all entities to be bootstrapped and in some cases, arbitrary translation can be applied to them by the use of so called “Translation tables” [SGH⁺15].

Performing ontology bootstrapping and the associated formal rules, the output will be an ontology (of a desired format, since the rules are stated in an abstract, system-independent way) and a set of RDF triples usable by R2RML to link the ontology to the source data (see Section 2.1.1). This RDF triples then can be used to implement either a materialized or a virtual OBDA approach (see Section 2.1.2).

For a description that goes beyond the following explanation, refer to [SGH⁺15]. A serialization format for OBDA specifications is introduced in Chapter 4 – [The OBDA SPECIFICATION LANGUAGE \(OSL\)](#). For the automatic bootstrapping of OBDA specifications, refer to Section 3.2.3.

3.2.1. Structure of OBDA specifications

An OBDA specification consists of several so-called “maps”, which are data records containing data and references to each other describing parts of the OBDA specification in statically defined fields [SGH⁺15]. For different aspects of the specification, there are different map types, while several maps typically exist for each type. Namely, these are *Entity maps* describing database tables, *Identifier maps* describing database primary keys, *Attribute maps* describing database columns, *Relation maps* describing database foreign keys, *Subtype maps* describing “is-a” relationships in the data and *Translation tables* describing desired translations of data.

The fields of the several types of maps and their interconnection via references is shown in Figure 3.2. Here, each field specifies, in that order, the field label, the field’s long name, the bootstrapping steps in which the field is used and the field’s short name. Fields storing a set of values have both their short and their long name suffixed with “. . .”. Note that each reference between two fields is denoted with a short field name contained in the source of the reference, specifying the field in which the reference is stored. What the values of the fields of the several types of maps express exactly and how they can be used is described in Section 3.2.2.

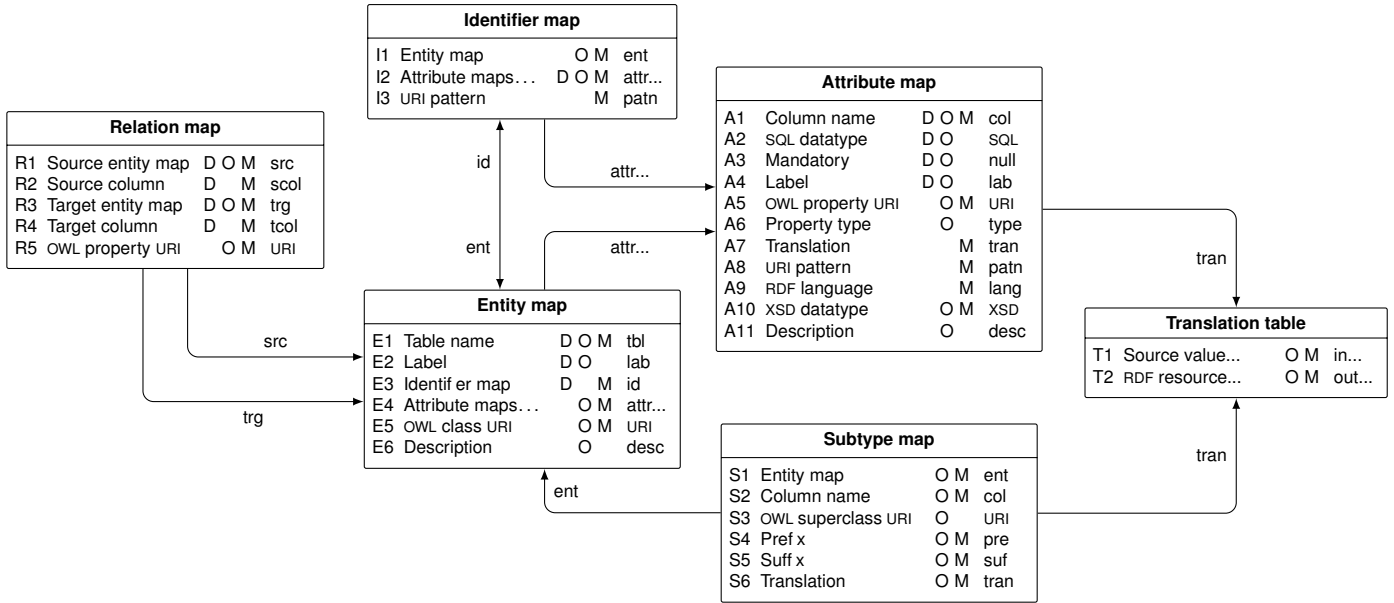


Figure 3.2.: Constitution of an OBDA specification. “→” means “references”. (from [SGH+15])

Entity maps, Identifier maps, Attribute maps and Relation maps directly relate to database concepts and each of them describes exactly one database table, primary key, column or foreign key, respectively, and vice versa. Subtype maps and Translation tables, on the other hand, represent concepts of the bootstrapping process or data to be added to the target ontology and are somewhat harder to obtain: Subtype maps represent “is-a” relationships in the target ontology to be determined from the source data [SGH+15] – heuristically or semi-automatically –, while Translation tables allow for the transformation of data values, for example from `TRUE` to `true` or from `No` to `false` [SGH+15]. Therefore, they also have to be determined heuristically or semi-automatically from the source data – considering the database schema only is not sufficient [SGH+15]. Note that, because of this, special care has to be taken to keep the maps synchronized with the data in case of Subtype maps and Translation tables.

The structural description of OBDA specifications in [SGH+15] does not propose a serialization format in which OBDA specifications can be stored or read and written by software and human agents. How this can be done is subject to this thesis, which introduces the OBDA SPECIFICATION LANGUAGE (OSL) designed exactly for this purpose in Chapter 4. How OBDA specifications can in turn be automatically bootstrapped, excluding Subtype maps and Translation tables, is described in Section 3.2.3.

3.2.2. Using OBDA specifications

As described in in Section 2.1.3 – OBDA specifications, using OBDA specifications provides several benefits when concerned in ontology bootstrapping. Principally, information about the bootstrapping process is collected in one place and can be used to manage the tools involved. This includes the availability of the URIs to be used in the constructed ontology from a central place, which is a great advantage, since URIs are central to an ontology [UG+96]. Additionally, all information on the database schema of the source database is available. Using Translation

tables, all these information can at will be made subject to transformations normalizing or correcting the data changing the database [SGH⁺15].

By the use of a single specification language like the OBDA SPECIFICATION LANGUAGE (OSL) to store OBDA specifications, the expense of converting between different data formats can be reduced significantly: assumed that there are n different formats to be handled with no means provided to convert between them, it is trivially deduce that the converting costs decrease from $\mathcal{O}(n^2)$ to $\mathcal{O}(n)$ by introducing a single central language.

Besides the structure of OBDA specifications, described in Section 3.2.1, Skjæveland et al. introduce a set of formal rules defining the bootstrapping process and the mapping of the source data to the generated ontology [SGH⁺15]. Using an OBDA specification, tools implementing these rules can easily and in a well-defined manner bootstrap an ontology and a mapping from the source data onto this ontology (or duplicate that source data, if the materialized OBDA approach is used, see Section 2.1.2). The mapping rules produce RDF triples which can be interpreted by R2RML to establish the mapping (for a short description of R2RML, see Section 2.1.1 – Basic concepts). They are accompanied by SQL statements specifying the queries over the source database used to link the ontology to the data. If the data source is not a relational database but another form of structured data, like CSV files, a database schema can be bootstrapped first by applying additional “database rules” [SGH⁺15]. Afterwards, the proceeding can continue as if the data source were a database, so this case is neglected in the following.

The rest of this section contains description of the information contained in the various types of OBDA specification maps and how it is used during bootstrapping, based on the description in [SGH⁺15]. Keep in mind that “bootstrapping” here refers to the ontology bootstrapping process specified by the OBDA specification, yielding a target ontology and mappings relating it to the source database – it does not refer to the bootstrapping of the OBDA specification itself. The text is meant to give a brief explanatory overview over the bootstrapping process using OBDA specifications. Thus, it focuses on the information they provide and how they are used to link the bootstrapped ontology to the source data, leaving out the SQL statements to be used to gain the datasets. How exactly the ontology is created is also left out, since this would have involved introducing too many technical details, and moreover, the topic is also comprehensible by describing the mapping only. For a detailed description of the bootstrapping process, including ontology creation and all formal rules to be applied, see [SGH⁺15]. For an explanation about how an OBDA specification containing Entity maps, Identifier maps, Attribute maps and Relation maps can be bootstrapped from a relational database schema, see Section 3.2.3. For details on URI generation, refer to Section 3.3 – Generating unique IRIs for OBDA specification map fields.

Entity maps

Entity maps provide information about the tables contained in the source database or in the intermediate database schema to be constructed, if the data source is not a database but some other source of structured information (see Section 2.1.2 – Ontology-based data access (OBDA)). The information provided by an Entity map includes the table name, a label describing the table and a (more detailed) description of the table. Furthermore, each Entity map references an Identifier map representing its primary key and a set of Attribute maps

representing its columns. Finally, an Entity map provides an OWL class URI identifying the represented table uniquely in the resulting ontology. As the name suggests, this URI is given to an OWL class which serves as OWL type (or more precisely: `rdf:type`) for all OWL individuals representing the datasets from the respective table in the target ontology (see Paragraph “[Identifier maps](#)” later in this section).

Supposing, for example, that an Entity map for a table “persons” provides the OWL class URI “mydb:persons”. Then, in the target ontology, all OWL individuals representing rows in the “persons” table, will be of `rdf:type` “mydb:persons”. If a data record in the “persons” table has the identifying URI pattern “mydb:person/{pno}” (see Paragraph [Identifier maps](#) later in this section), this type information will be expressed by the following RDF triple:
mydb:person/{pno} rdf:type mydb:persons.

Identifier maps

Identifier maps describe database primary keys contained in the source database or in the intermediate database schema to be constructed, if the data source is not a database (see Section 2.1.2). Each Identifier map contains a reference back to the respective Entity map, representing the database table the primary key belongs to. Furthermore, each Identifier map references a set of Attribute maps, representing the database columns the primary key consists of (see next paragraph). Finally, an Identifier map provides a URI pattern, allowing OWL individuals in the bootstrapped target ontology that represent datasets to be identified. A URI pattern contains placeholders like “{ $\$1$ }” for all primary key columns, which are replaced with the respective column names, surrounded by curly braces, during the bootstrapping process, to yield a valid R2RML template [[W3CR12b](#)]. Since the column name substituted in is still a placeholder, the result of this substitution is still a URI pattern and not a URI. Since such a URI pattern uniquely identifies a dataset from a given database table – when data values are substituted in –, it will be called *identifying URI pattern* in the following.

Consider the URI pattern “mydb:person/{ $\$1$ }”. Replacing the placeholder “{ $\$1$ }” with the column name “pno” in curly braces yields the following identifying URI pattern:
“mydb:person/{pno}”.

Attribute maps

Attribute maps provide information about database columns contained in the source database. Each Attribute map carries the column name, information whether having a value in this column is mandatory for a dataset (in SQL terms: whether it has a `NOT NULL` constraint), a label describing the column as well as an extended description of the column. A database column is represented as a relation in the final ontology, thus, in OWL terms, as an `owl:DataProperty` or an `owl:ObjectProperty`. The Attribute map provides the URI for this OWL property.

Additionally, it specifies the datatype of values in the represented column in the following manner: Three fields are provided for this purpose, `SQL datatype`, `RDF language` and `XSD datatype`. If the `XSD datatype` field is nonempty, its value is specified to be the datatype for values in the column the Attribute map represents (note that OWL only knows XSD datatypes [[W3C12](#)]). Otherwise, if the value of the `SQL datatype` is a standard SQL type, it will be mapped to an XSD datatype and the resulting type is specified as datatype for values

in the respective column. If neither of the above is the case and the `RDF language` field is nonempty, values in the respective column will be interpreted as strings with the value of the `RDF language` field applied as RDF language tag (see [W3C14b]). If neither of the above is the case, values in the respective column will be interpreted as strings without an RDF language tag.

Finally, an Attribute map specifies whether the column shall be represented as an `owl:DataProperty` (for non-foreign-key columns) or as an `owl:ObjectProperty` (for foreign key columns). The field specifying that – `Property type` – also allows, as a further distinction of object properties, whether the property’s target URI should be the column name placed in a URI pattern provided by the Attribute map, similarly to URI patterns in Identifier maps (see example at the end of this paragraph), or if it shall simply be the column name, possibly with a translation from a Translation table, specified by the Attribute map, applied. The former option is useful for example when using custom property URIs to express relations between source data and the target ontology. If an `owl:DataProperty` is generated, it always has the column name as target URI, without the use of a URI pattern. Note that it is sufficient to have the column name be the target of the property, since only a *mapping* to the source data is generated.

Consider an Attribute map representing a column named “name” to be mapped to an `owl:DataProperty`. Supposing the OWL property URI the Attribute map specifies is “`mynamespace:lastName`” and each dataset containing the name column has the identifying URI pattern “`mydb:person/{pno}`” (see Paragraph “Identifier maps”). Then, during the bootstrapping process the following RDF triple will be produced:

```
mydb:person/{pno} mynamespace:lastName "{name}"
```

This triple can easily be interpreted by R2RML, which on request then retrieves the queried name from the data source.

Consider, as a more elaborate example, an Attribute map representing a column named “company” and to be mapped to an `owl:ObjectProperty` with the use of the URI pattern “`http://otherdb/{$1}`”. Supposing the OWL property URI the Attribute map specifies is “`mynamespace:hasSameOwnerAs`”, there is no datatype specified and each dataset containing the company column has the identifying URI pattern “`mydb:company/{cmpno}`” (see Paragraph “Identifier maps”). Then, the following RDF triple will be produced during the bootstrapping process:

```
mydb:company/{cmpno} mynamespace:hasSameOwnerAs http://otherdb/{company}
```

Note that this only makes sense if R2RML can expand “`http://otherdb/{company}`” to a valid subject for each value in the company column of the database, and if all rows in the respective database table are indeed entities having the same owner as the company specified in the company column.

Relation maps

Relation maps represent foreign keys contained in the source database. Each Relation map references the Entity maps representing the foreign key’s child table and parent table, respectively. Furthermore, it provides the column names of both the foreign key columns and the referenced columns and specifies an OWL property URI. Relation maps allow the relations expressed in the source data via foreign key relationships to be included into the bootstrapped

ontology. This happens in a simple and straight-forward manner: for each foreign key relationship, exactly one triple is generated which contains the two identifying URI patterns representing the source and the target dataset of the foreign key, respectively, and the OWL property URI specified by the Relation map.

Supposing, for example, that a Relation map expressing a relation between datasets with the identifying URI patterns (see Paragraph “[Identifier maps](#)”) “mydb:persons/pno/{pno}” and “mydb:companies/cmpno/{cmpno}”, respectively, and specifying the OWL property URI “mynamespace:isEmployedAt”. This will result in the following RDF triple to be generated during the bootstrapping process:

```
mydb:persons/pno/{pno} mynamespace:isEmployedAt mydb:companies/cmpno/{cmpno}
```

Subtype maps

Subtype maps provide a means to automatically add subclass-superclass relationships to the target ontology during the bootstrapping process. They specify an Entity map and a column name defining a table and a column, respectively, that exist in the source database and contain the values to be declared as belonging to the subclass. Furthermore, they store a prefix, a suffix and possibly a reference to a Translation table which are used to generate a URI for that subclass. Finally, they provide the URI of the superclass. This can be, for instance, some OWL class being created during the bootstrapping process or already existing in some imported ontology. The URI generated for the subclass contains the data value of the respective database column, thus every dataset gets its own (sub)class. During bootstrapping, an RDF triple declaring the value to belong to that subclass is generated for each data value of the respective table column in the source database. The limitation to the desired value only thereby happens by a restriction on the SQL statement accompanying the respective triple, not by limiting the triple to only cover a specific data value. The actual subclass-superclass relationship is expressed during the creation of the ontology. Note the difference from the previously described mapping rules, which produced triples independent from the data values in the source database.

Consider a Subtype map specifying a table column containing the values “Purchase” and “Sales” with the datasets having the identifying URI pattern (see Paragraph “[Identifier maps](#)”) “mydb:managers/mno/{mno}”. Supposing, the Subtype map specifies the prefix “mydb:manager/of_department/”, no suffix, no translation table and the supertype URI “mynamespace:persons”. This will result in the generation of the following triples during the bootstrapping process:

```
mydb:managers/mno/{mno} rdf:type mydb:manager/of_department/Purchase
```

```
mydb:managers/mno/{mno} rdf:type mydb:manager/of_department/Sales,
```

while “mydb:manager/of_department/Purchase”

and “mydb:manager/of_department/Sales” will be subclasses of class

“mynamespace:persons” in the target ontology. The accompanying SQL statement will ensure that, despite the use of the R2RML template “mydb:managers/mno/{mno}”, not *every* manager will be declared as the manager of *every* department.

Translation tables

Translation tables allow for transforming URIs or other strings in arbitrary ways, by simply mapping each string to be translated to a target string.

They don't reflect in the target ontology in any form but are used only during the bootstrapping process.

3.2.3. Bootstrapping OBDA specifications

How OBDA specifications can in turn be bootstrapped from database schemata is subject to this thesis and is explained in this section. For the description of the software developed to automate this, see Chapter 5 – [The DB2OSL software](#). The description in this section assumes an SQL database as data source. However, ontology-based data access and OBDA specifications are not limited to SQL databases, as mentioned in Section 2.1.2 – [Ontology-based data access \(OBDA\)](#) and in Section 3.2.2 – [Using OBDA specifications](#). Furthermore, this section refers to OBDA specifications without assuming any specific format in which they are represented. How OBDA specifications are represented internally by the DB2OSL software, is described in Section 5.5.4 – [Fine structuring of DB2OSL](#). For the description of a format to serialize OBDA specifications – the output format of DB2OSL –, refer to Chapter 4 – [The OBDA SPECIFICATION LANGUAGE \(OSL\)](#).

Subtype maps and Translation tables are not considered in this approach, since they cannot be bootstrapped from schema information only but have to be determined from the input data (see Section 3.2.1 – [Structure of OBDA specifications](#)). Thus, the bootstrapped OBDA specification does not contain maps of these types. Including them is a significant challenge in its own right and, since the use of heuristics or user decisions would be necessary, would make the process involve human supervision at least. Apart from that, the bootstrapping is an easy and straight-forward task which can be carried out fully automatic [[MNS03](#)].

Recall the structure of an OBDA specification explained in Section 3.2.1 – [Structure of OBDA specifications](#). The map types considered in this approach are Entity maps, Attribute maps, Identifier maps and Relation maps. The assignment of values to their fields is summarized in Table 3.1, only hinting at how maps are generated. Both the generation of the maps and the assignment of values to their fields are described in the rest of this section, with one exception: since the generation of URIs (or IRIs) in the context of OBDA specifications is an essential topic which requires some conceptual efforts, it is described in a separate section, Section 3.3.

Map type	Field name	Value
Entity map	Table name	SQL table name
Entity map	Label	<empty>
Entity map	<i>Identifier map</i>	Identifier map for table
Entity map	<i>Attribute maps...</i>	Attribute maps for table columns
Entity map	OWL class URI	URI(table)
Entity map	Description	SQL table description
Identifier map	<i>Entity map</i>	Entity map for corresponding table
Identifier map	<i>Attribute maps...</i>	Attribute maps for primary key columns
Identifier map	URI pattern	URIpattern(table)
Attribute map	Column name	SQL column name
Attribute map	SQL datatype	SQL datatype of column
Attribute map	Mandatory	SQL NOT NULL property of column (true or false)
Attribute map	Label	<empty>
Attribute map	OWL property URI	<empty> for foreign key columns, else URI(table, column)
Attribute map	Property type	“ ObjectProperty ” for foreign key columns, else “ DataProperty ”
Attribute map	<i>Translation</i>	<empty>
Attribute map	URI pattern	<empty>
Attribute map	RDF language	<empty>
Attribute map	XSD datatype	<empty>
Attribute map	Description	SQL column description
Relation map	<i>Source entity map</i>	Entity map for foreign key child table
Relation map	Source column	Foreign key child columns (SQL column names)
Relation map	<i>Target entity map</i>	Entity map for foreign key parent table
Relation map	Target column	Foreign key parent columns (SQL column names)
Relation map	OWL property URI	URI(table, foreignKey)

Table 3.1.: Assignment of values to fields of OBDA specification maps

Entity maps

Exactly one Entity map and one Identifier map is generated per table contained in the source database. The generated Identifier map is referenced by the Entity map’s **Identifier map** field. Similarly, exactly one Attribute map is generated per table column and these Attribute maps are referenced by the Entity map’s **Attribute maps...** field. The Entity map’s **Table name** field is set to the SQL name of the table, the **Label** field remains empty. A URI identifying the table is generated and stored in the Entity map’s **OWL class URI** field. The SQL table description is copied into the Entity map’s **Description** field.

Identifier maps

An Identifier map represents exactly one primary key in the source database and is referenced by the Entity map representing the table containing the primary key constraint. In addition, it references this table in its **Entity map** field, so that there is a bidirectional referencing. The Attribute maps representing the columns constituting the primary key are referenced by the Identifier map's **Attribute maps...** field. A URI pattern, allowing datasets (thus, rows in the source database) to be identified in the target ontology, is generated and put in the Identifier map's **URI pattern** field.

Attribute maps

An Attribute map represents exactly one column in the source database and is referenced by the Entity map representing the table containing the column. The Attribute map's **Column name** field is set to the SQL column name of the column, the **SQL datatype** field is set to its SQL datatype. The **Mandatory** field is set to **true** if the column has the SQL NOT NULL constraint, otherwise to **false**. If the column is part of a foreign key, the **OWL property URI** field remains empty. Otherwise, a URI identifying the column is generated and stored in the Attribute map's **OWL property URI** field. The **Property type** field is set to "ObjectProperty" if the column is part of a foreign key, otherwise to "DataProperty". The SQL column description is copied into the Attribute map's **Description** field. The remaining columns, **Label**, **Translation**, **URI pattern**, **RDF language** and **XSD datatype** remain empty.

Relation maps

A Relation map represents exactly one foreign key in the source database. It contains fields storing the parent and child table of the foreign key: the **Source entity map** field, referencing the Entity map representing the child table of the foreign key, and the **Target entity map** field, referencing the Entity map representing the parent table of the foreign key. The SQL column names of the foreign key columns (thus, column names in the child table) are copied into the **Source column** field of the Relation map, and the SQL column names of the referenced columns in the parent table are copied into its **Target column** field. Note that, in contrast to Identifier maps representing primary keys, it is not referenced by any Entity map (or any other map).

3.2.4. Conclusion on OBDA specifications

To be mentioned first, the author lacks experience in the matter of performing ontology bootstrapping practically and therefore, the helpfulness of the approach in real-world applications cannot truly be judged.

However, the approach appears very promising. The concentration of relevant information in one place is definitely useful when dealing with changing environments and large amounts of data. The theoretical benefits of OBDA specifications are convincing (see Section 2.1.2 – [Ontology-based data access \(OBDA\)](#)) and especially the degree of extendability, allowing for

arbitrary URIs and arbitrary translations of URIs and data values and thus the link to existing ontologies is auspicious.

It may, however, be a possible improvement if date stamps or other versioning information and similar administrative information could be included in an OBDA specification, to face the challenge of large amounts of changing data even better.

3.3. Generating unique IRIs for OBDA specification map fields

As explained in Section 2.1.1 – [Basic concepts](#), IRIs play a central role in diverse topics related to ontology-based data access. They provide the means to uniquely identify entities, which of course is a necessity for data retrieval. As also explained in Section 2.1.1, every URI is also an IRI, so although Skjæveland et al. use the term “URI” in the introduction of their approach of using OBDA specifications for ontology bootstrapping – and that term is also used in Section 3.2, which describes this approach – in this section the general term “IRI” is used, marking that the introduced concepts are valid for all types of IRIs.

When dealing with ontology bootstrapping using OBDA specifications, it is important to differentiate between the three types of IRIs occurring in this matter, which will be underlined by the following unambiguous naming:

- *Data IRIs* identify entities in the bootstrapped ontology
- *OBDA IRIs* are used as values for the fields of OBDA specification entities
- *OSL IRIs* identify components in serialized OBDA specifications, using the OBDA SPECIFICATION LANGUAGE (OSL) introduced in Chapter 4 for serialization

Skjæveland et al. do not define or assume a particular scheme for IRI generation in their introduction of OBDA specifications [[SGH⁺15](#)]. Instead, the IRI generation strategy is only adumbrated by giving examples of entities having IRIs. The exemplified scheme was used for the implementation of the DB2OSL software bootstrapping OBDA specifications from relational database schemata, which is described in this thesis (see Section 5 – [The DB2OSL software](#) and Section 6 – [Implementation of DB2OSL](#)). The direct mapping approach for ontology bootstrapping described in Section 3.1, on the other hand, introduces a scheme for IRI generation [[W3CR12a](#)], but with this scheme, name clashes can occur, as explained in Section 3.1.3. The OBDA SPECIFICATION LANGUAGE (OSL), finally, defines a proper scheme for OSL IRIs, as is explained in Section 4.2 – [Design of the OBDA SPECIFICATION LANGUAGE \(OSL\)](#).

To prevent confusion, it shall be noted that OBDA IRIs are closely related to Data IRIs – all OBDA IRIs occur similarly or unchanged in the ontology resulting from the bootstrapping. However, because they are not always

In the following, an enhanced scheme for the generation of OBDA IRIs is proposed, which resembles the scheme used for OSL IRIs and which also may serve as a blueprint for other IRI generation strategies.

3.3.1. Requirements for the IRI scheme

As explained in Section 2.1.1 – [Basic concepts](#), the main requirement on an IRI generation scheme is uniqueness of the IRIs: no two entities must be possibly assigned the same IRI, regardless of their kind, of how low the probability of a name clash (IRI collision) is or of the conditions leading to a name clash. Additionally, IRI uniqueness shall be independent from the base IRIs, thus a base IRI shall be arbitrarily selectable for each generation process without introducing name clashes even with IRIs having other base IRIs.

As to OBDA specification entities, the following kinds of IRIs have to be available, including IRI patterns:

- Entity map OWL class IRIs
- Identifier map IRI patterns
- Attribute map OWL property IRIs
- Attribute map IRI patterns
- Relation map OWL property IRIs
- Subtype map (IRI) prefixes
- Subtype map (IRI) suffixes
- Subtype map OWL superclass IRIs

As Subtype map OWL superclass IRIs are IRIs of data entities already existing in the target ontology by some means (see Section 3.2.2 – [Using OBDA specifications](#)), they do not have to be generated and thus are ignored in the following. Exactly the same holds for Attribute map IRI patterns. Furthermore, this approach creates Subtype map IRI prefixes already leading to unique IRIs for Subtype map subclasses and so Subtype map IRI suffixes are ignored in the following. Since an IRI generation scheme cannot avoid collisions with existing IRIs out of its outreach and these collisions can easily be prevented, for example, by giving them another base IRI (see Section 2.1.1 – [Basic concepts](#)), this case is excluded from the requirement that no two URIs must collide under any circumstances. However, the user shall be able to chose such externally generated IRIs from an infinite set of IRIs, while being sure that no name clashes will occur.

So compendious, the requirements on the IRI generation scheme are that Entity map OWL class IRIs, Identifier map IRI patterns, Attribute map OWL property IRIs, Relation map OWL property IRIs and Subtype map IRI prefixes can be generated that, regardless of the chosen base IRIs, don't clash among another, while leaving an infinite set of predictable IRIs that don't clash with any of the generated IRIs.

3.3.2. Avoiding name clashes in the IRI scheme

This section mentions the occasions of name clashes (IRI collisions) and the measures to avoid them in a rather technical and verbose way, using one paragraph for each type of IRIs, IRI patterns and IRI prefixes. To avoid unnecessary obfuscation, the term “IRIs” is used in many places where, precisely, “IRIs, IRI patterns and IRI prefixes” is meant.

Generating unique Entity map OWL class IRIs ignoring base IRIs is not much of a problem, assuming database table names are distinct, which is guaranteed in a common database system like SQL [VDL89]. Including the table name into an Entity map OWL class IRI is sufficient to prevent it from colliding with other IRIs with the same base IRI. However, when taking two different base IRIs into account that are used for two IRIs created according to this scheme, things get more complicated.

Consider, for example, a database table named “Persons” and a table named “Persons_TABLE_Persons”. Generating an IRI according to the scheme “<base:>TABLE__<table name>” for each of these tables, using the base IRI “TABLE__Persons__” for the first one and the empty base IRI for the second one, both tables will get the IRI “TABLE__Persons__TABLE_Persons”, although the table name was included into the IRI in both cases. The problem is that the “TABLE__” string occurring in the table name cannot be discriminated from the “TABLE__” string added in the course of IRI generation or the “TABLE__” string occurring in the base IRI. To solve the problem, a marker has to be included in the URI which definitely indicates the beginning of the table name. In addition, this marker will uniquely identify Entity map OWL class IRIs. For both aims to be achieved, an escape symbol must be used, which makes the marker unique at least outside the base IRI part, by escaping the marker whenever it occurs in the table name.

Regarding Identifier map IRI patterns, the IRI resulting from the expansion of the pattern will contain the column names of the primary key represented by the respective Identifier map [SGH⁺15]. Further on, the table name of the database table containing that primary key has to be included in the IRI pattern, since two distinct tables may have primary keys with equally named columns. This will make the IRI pattern a unique Identifier map IRI pattern, since a database table can be assumed to only have one primary key, as is the case in common database systems like SQL [VDL89]. The fact that primary key values are unique for each dataset ensures that unique Identifier map IRI patterns expand to unique IRIs. Moreover, it has to be ensured, that Identifier map IRI patterns do not collide with IRIs of other kinds. Taking arbitrary and particularly varying base IRIs into account, a definite marker has to be included in the IRI pattern and other occurrences of this marker in the IRI pattern have to be escaped. This uniquely identifies Identifier map IRI patterns and unambiguously distinguishes the table name from the rest of the IRI pattern.

Concerning Attribute map OWL property IRIs, they will be unique among their kind when they include the column name of the database column they represent besides the table name of the table containing it, since database table names can be assumed to be distinct and column names can be assumed to be unique within a table, which is guaranteed in a common database system like SQL [VDL89]. Furthermore, Attribute map OWL property IRIs have to be prevented from colliding with IRIs of other kinds.

Taking arbitrary and particularly varying base IRIs into account, definite markers have to be included in the IRI and other occurrences of this marker in the IRI have to be escaped. This uniquely identifies Attribute map OWL property IRIs and unambiguously distinguishes the table name and the column name from the rest of the IRI and from one another.

Regarding Relation map OWL property IRIs, including the table name and the column names of both the foreign key represented by the Relation map (or the containing table, respectively) and the referenced key (or its containing table, respectively) in the IRI will make it a unique Relation map OWL property IRI. Note that including only the table name and the column

names of the foreign key (or its containing table, respectively) would not be sufficient, since several distinct foreign keys covering exactly the same columns can exist in a table (this is what the IRI generation scheme of the direct mapping approach misses). The same applies of course for the referenced table and its columns – several foreign keys can reference them. Moreover, these Regarding Relation map OWL property IRIs have to be prevented from colliding with IRIs of other kinds.

Taking arbitrary and particularly varying base IRIs into account, definite markers have to be included in the IRI pattern and other occurrences of this marker in the IRI have to be escaped. This uniquely identifies Relation map OWL property IRIs and in particular their parts providing the table and column names.

Concerning Subtype map IRI prefixes, they must include the column name of the database column containing the values to be declared belonging to the subclass. Further on, since another database table could contain a column of the same name, the IRI must include the table name of the database table containing the column. This will make Subtype map IRI prefixes unique among their kind. Note that a Subtype map IRI prefix, similarly to an IRI pattern, does not specify the final IRI but is subject to expansion. This expansion can yield the same IRI for different data records, which, however, is not considered a collision, since this behavior is intentional: every two data records having the same value in the respective column – and only those – will get the same IRI. Additionally, it has to be ensured, that Subtype map IRI prefixes do not collide with IRIs of other kinds.

Taking arbitrary and particularly varying base IRIs into account, definite markers have to be included in the IRI pattern and other occurrences of this marker in the IRI prefix have to be escaped. This uniquely identifies Subtype map IRI prefixes and unambiguously distinguishes the table name and the column name from the rest of the IRI prefix and from one another.

For an example that makes awkwardly – or fraudulently – chosen base IRIs introduce name clashes, see the paragraph about Entity map OWL class IRIs at the beginning of this section.

3.3.3. The proposed IRI generation scheme

This section introduces an IRI generation scheme meeting the requirements formulated in Section 3.3.1.

In this section, the following strings are subsumed under the term *marking strings*: “TABLE__”, “TBL__”, “PROP__”, “REF__” and “SUBTYPE__”.

The string built by escaping (prefixing) all occurrences of marking strings or ‘~’ characters in a string *s* with a ‘~’ character will be called the *IRI-safe version* of *s*.

The IRI generation scheme is presented in Table 3.2. Here,

<base:> refers to the base IRI to be used for the generated IRI (see Section 2.1.1 – [Basic concepts](#)),

<cl. tbl name> refers to the table name of the database table concerning (see Section 3.3.2) in its IRI-safe version,

<cl. name 1st pk col> refers to the name of the first primary key column concerning (see Section 3.3.2) in its IRI-safe version,

</...> refers to the continuation of the previous pattern using the remaining primary key or foreign key columns,

<cl. col name> refers to the name of the column in question (see Section 3.3.2) in its IRI-safe version,
 <cl. src tbl> refers to the table name of the database table containing the respective foreign key (see Section 3.3.2) in its IRI-safe version,
 <cl. 1st src col> refers to the name of the first foreign key column of the respective foreign key (see Section 3.3.2) in its IRI-safe version,
 <cl. tgt tbl> refers to the table name of the table referenced by the respective foreign key (see Section 3.3.2) in its IRI-safe version and
 <cl. 1st tgt col> refers to the name of the first column referenced by the respective foreign key (see Section 3.3.2) in its IRI-safe version.

IRI type	Proposed IRI
Entity map OWL class IRI	<base:>#TABLE__<cl. tbl name>
Identifier map IRI pattern	<base:>#TBL__<cl. tbl name>/<cl. name 1st pk col> /{\$1}/</...>
Attribute map OWL property IRI	<base:>#PROP__<cl. tbl name>__<cl. col name>
Relation map OWL property IRI	<base:>#REF__<cl. src tbl>/<cl. 1st src col> </...>/<cl. tgt tbl>/<cl. 1st tgt col></...>
Subtype map IRI prefixes	<base:>#SUBTYPE__<cl. tbl name>__ <cl. col name>/

Table 3.2.: Proposed IRIs to be used in OBDA specification map fields. Spaces were inserted for readability.

Note that the IRI schema enforces only the use of ASCII characters, which makes it suitable in particular for URIs, which allow only ASCII characters.

It is easily verified that the proposed IRI scheme is correct regarding the requirements described in Section 3.3.1: it provides unique IRIs for all types of IRIs it allows to create, regardless of the chosen base IRI (see proof in Section 3.3.4). Furthermore, the IRI scheme is expressive: ignoring the base IRI part, the kind of entity identified by the IRI can be determined by beginning of the IRI. In addition, it is regular in that the name of the containing table always occurs before the name of the first database column.

Taking the information in Section 3.3.2 into account, it is trivial to observe that the suggested IRI scheme, leaving out the demand of IRI-safe versions, is still correct, given that all IRIs are generated using the same base IRI. When allowing varying base IRIs, however, the IRI-safe versions are necessary, since they allow the discrimination of a marking string included by the IRI generation scheme from a marking string contained in the name of a database entity, even with different base IRIs.

Note that it is in any case necessary that the beginnings of all kinds of IRIs be mutually different: if, for example, an Identifier map IRI pattern also would commence with “<base:>#TABLE__”, a table named “PERSONS/{17}” (which is a valid table name for example in SQL [VDL89]) possibly could get an Entity map OWL class IRI assigned which clashes with the IRI resulting from the expansion of the IRI pattern “<base:>#TABLE__PERSONS/{\$1}”.

Note that furthermore, it is necessary to include the rhombus sign (‘#’) in the IRI: if this would not be done, a base IRI having ‘~’ as its last character would make the marking string

indistinguishable from a marking string escaped with a ‘~’ character because it occurs, for instance, in a table name.

3.3.4. Proof of correctness of the proposed IRI scheme

As described in Section 3.3.1, the IRI scheme described in Section 3.3.3 is required to generate several types of IRIs without introducing name clashes, thus two equal IRIs for two distinct entities, independently of the chosen base IRIs. Additionally, the user shall be able to chose additional IRIs he can be sure won’t collide with IRIs generated with the scheme from an infinite set.

In this proof, like in Section 3.3.2, the strings “TABLE__”, “TBL__”, “PROP__”, “REF__” and “SUBTYPE__” are called *marking strings*.

Strings prefixed by ‘~’ are referred to as *escaped*, while strings not prefixed by ‘~’ are referred to as *unescaped*.

In the following, it is proven that the IRIs of each type do not clash, neither among themselves nor with IRIs of other types. Since all generated IRIs begin with a marking string, every IRI *not* beginning with a marking string, thus an infinite quantity, is sure not to collide with any of the generated IRIs, and so, the correctness regarding to the stated requirements is then proven.

Each Entity map OWL class IRI (including its base IRI) is of the form α TABLE__ β , with α ending with ‘#’ and β and not containing any unescaped marking strings.

Thus, the “TABLE__” string before β is the last unescaped marking string in the IRI and so, β is the table name. This makes the IRI unique among all Entity map OWL class IRIs (see considerations in Section 3.3.2).

Furthermore, IRIs of other types cannot have “TABLE__” as the last unescaped marking string and so the IRI cannot collide with an IRI of another type.

Thus, the IRI is indeed unique.

The proof for Identifier map IRI patterns, Attribute map OWL property IRIs, Relation map OWL property IRIs and Subtype map IRI prefixes is exactly analogous.

□

3.3.5. Conclusion on the IRI generation scheme

As the scheme fulfills the requirement of generating unique IRIs while leaving the user an infinite amount of predictable IRIs guaranteed *not* to be generated, it is definitely useful. Through the sole requirement of ASCII characters to be available, it is also widely applicable (for example for URIs). Furthermore, it produces IRIs which are – as far as possible – readable and regular (note, for example, that the first variable string, besides the base IRI, is always a table name). Finally, it is simple and predictable. Generally, it is suited very well for the operation in the area of ontology bootstrapping, where readability and guaranteed uniqueness is important and predictability and simplicity is a big advantage (for example this can loose coupling of components and generally ease the implementation of a system).

A drawback from focusing on these aspects, however, is the tallness of the resulting IRIs.

4. The OBDA Specification Language (OSL)

This section introduces the OBDA SPECIFICATION LANGUAGE (OSL), a language to store and exchange OBDA specifications (which are described in Section 3.2). Section 4.1 describes the requirements and objectives for the language. Section 4.2 explains the design of the language. Section 4.3 contains the formal specification of OSL. Section 4.4 includes an example of an OSL specification. Finally, Section 4.5 contains a conclusion on how good the stated objectives were met.

OSL is the output format of the DB2OSL software, described in Chapter 5 – [The DB2OSL software](#) and in Chapter 6 – [Implementation of DB2OSL](#).

4.1. Requirements on the OBDA Specification Language (OSL)

As the OBDA SPECIFICATION LANGUAGE is to be used by diverse parties, including computer programs, it is extremely important that it is well-defined, thus that there exists a carefully written, precise and complete specification (this specification is provided in Section 4.3). Furthermore, it must be easily learnable for new users and easy to implement, while at best being supported by existing tools. Additionally, it must cleanly fit into the environment of ontologies, being readable for humans so that modifications are easy to accomplish and debugging is easy.

Also, it must be free from copyright or other legal restrictions, must be saveable to a data file and must allow for exposition in the web.

Besides, the concepts and the specification should be simple and comprehensible.

4.2. Design of the OBDA Specification Language (OSL)

The first and most important decision was to make the OBDA SPECIFICATION LANGUAGE be a plain text format. This way, the requirements of storing them in data files and exposing them in the web are trivially achieved.

As described in Section 3.2.1 – [Structure of OBDA specifications](#), an OBDA specification consists of several types of maps, all containing data entries and links to other maps. This fits perfectly into the environment of ontologies and OWL, with data properties being the obvious choice to represent contained data entries and object properties being the obvious choice to

represent links between maps. Also, OWL is free from copyright or similar restrictions [W3C12] and is known to most users in the field of ontology bootstrapping, who ideally have advanced knowledge of it and already have tools able to process it deployed, which they are also familiar with. Thus, it was decided to base the OBDA SPECIFICATION LANGUAGE on OWL.

Another advantage of this approach is that the specification is kept compact and focused on the entities that the language has to represent rather than primarily dealing with technical details. In particular, many of those details can be formulated as OWL restrictions in a header ontology demanded to be imported by documents conforming to the OSL specification. Thus, they are not only specified precisely but they are also stipulated in a machine-readable form for which tools are widely available, enabling the user to check many aspects of an OSL document for conformity with minimal effort.

Since OWL already defines several serialization formats [W3C12], the task of designing OSL reduces to construct a reasonable structure for OWL ontologies representing OSL specifications. Fortunately, this is mostly a simple and straight-forward task: every type as OBDA specification map is represented as an OWL individual, using OWL properties in the previously mentioned way to relate them to their data object and referenced objects. This way, tool support is given as is an easy implementation – supposing an OWL library as is available for many programming languages.

Furthermore, by using the XML serialization of OWL or a similar one, the resulting OSL document is human-readable and easy to modify. To even better achieve this aim, all OSL OWL entities (the ones belonging to the language itself rather than a specification) were put into a separate namespace and that namespace was recommended to be used as base namespace in the specification of the OBDA SPECIFICATION LANGUAGE.

Attempts were made to make the language flexible: Arbitrary individuals were allowed to exist besides the mandatory ones, as long as they do not have IRIs defined in the specification of OSL. This keeps open manifold possibilities for populating OSL specifications with additional information without producing non-conforming specifications. Additionally, the mechanism to represent lists of strings leaves room for placeholders or other algorithmic artifices or to simply save space: unlike most other OSL entities, string lists and their nodes are not identified by their IRIs but by references from the preceding node. This allows string lists to be shared by several OSL entities. Furthermore, it is allowed for a list node not to have a string entry, enabling implementors to create nodes which join several distinct lists without having to know whether and how the list continues.

As regards simplicity, consider Table 4.2 and Table 4.3, specifying the types (or more precisely: type IRIs) to which OSL individuals must belong and the IRIs to be used for OSL properties, respectively. All names exactly comply with the respective terminology in [SGH+15] (see also Section 3.2.1 – Structure of OBDA specifications), removing spaces, with one exception: the properties “osl:tt_sourceValues”, “osl:tt_targetValues”, “osl:rm_sourceColumns” and “osl:rm_targetColumns” have an ‘s’ appended, which underlines that the respective properties refer to multiple values. Indeed, these are precisely the properties referring to string lists, so a second irregularity was also accepted in their case, for the sake of having the plural form, which in these cases more precisely describes the expectable property value.

4.2.1. IRIs in the OBDA Specification Language (OSL)

As already mentioned in Section 4.2, OSL entities are usually identified and “found” by their IRIs. This has the advantage of being able to directly access them only by knowing the discriminating keywords occurring in their IRI, and of course the scheme by which the IRI is generated, which is a fixed scheme presented in Table 4.1) of the specification of the OBDA SPECIFICATION LANGUAGE. Conversely, when processing an OSL IRI its purpose and type can be determined solely from its IRI.

The scheme used in the OBDA SPECIFICATION LANGUAGE to generate IRIs resembles the scheme proposed to generate IRIs for OBDA specification map fields introduced in Section 3.3. It is, however, much simpler in that it does not have to deal with arbitrary base IRIs. It was quickly figured out, that the inclusion of unique “marking strings” that discriminate between each IRI type at a certain place (here, at the end) is sufficient to make the IRIs unique (refer to Table 4.1 to see them).

4.3. Specification of the OBDA Specification Language (OSL)

¹ An OSL document is a valid OWL 2 document (as described in [W3C12]) containing individuals and data that represent the OBDA specification, as well as OWL properties that connect them. The individuals and OWL properties are recognized and mapped to their roles by their IRIs.

² An OSL document may contain more OWL entities (with IRIs not defined in this specification), which are ignored.

³ An OSL document has to declare all individuals having different IRIs as different from each other (except those which are ignored, see Paragraph 2) of this specification. It is recommended to use the `owl:AllDifferent` OWL statement for this purpose.

⁴ Unless stated otherwise, IRIs mentioned in the following are IRIs relative to a base IRI chosen by the user being empty (which makes the IRIs absolute [W3C09]) or ending with a hash character (`#`).

It is recommended to use that base IRI as `xml:base` XML attribute.

IRIs prefixed with `osl:` are IRIs relative to the IRI

<http://w3studi.informatik.uni-stuttgart.de/~martispp/ont#> .

⁵ An OSL document has to import the following ontology (referred to as “the OSL header” in the following):

<http://w3studi.informatik.uni-stuttgart.de/~martispp/ont/db2osl.owl>

⁶ The OWL individuals described by the OSL document representing the certain types of OBDA maps must have the IRIs specified in Table 4.1 (for base IRIs, see Paragraph 4 of this specification). Here, `<class URI>` refers to the OWL `class` URI field of the respective entity map for entity maps, to the OWL `class` URI field of the associated entity map for identifier maps, to the OWL `class` URI field of the associated entity map for subtype maps and

Map type	OWL IRI
Entity map	<class URI>__ENTITY_MAP
Attribute map	<property URI>__ATTRIBUTE_MAP
Identifier map	<class URI>__IDENTIFIER_MAP
Relation map	<property URI>__RELATION_MAP
Subtype map	<class URI>__SUBTYPE_MAP
Translation table of attribute map	<property URI>__ATTRIBUTE_MAP__TRANSLATION_TABLE
Translation table of subtype map	<class URI>__SUBTYPE_MAP__TRANSLATION_TABLE

Table 4.1.: OWL individual IRIs in OSL

Map type	OWL class IRI
Entity map	osl:EntityMap
Attribute map	osl:AttributeMap
Identifier map	osl:IdentifierMap
Relation map	osl:RelationMap
Subtype map	osl:SubtypeMap
Translation table	osl:TranslationTable

Table 4.2.: OWL class membership of map representations in OSL

to the OWL `class` URI field of the entity map associated with the respective subtype map for translation tables of subtype maps.

Similarly, `<property URI>` refers

to the OWL `property` URI field of the respective attribute map for attribute maps (or, if it is empty, the value that would have been generated for it if it weren't empty),

to the OWL `property` URI field of the respective relation map for relation maps and

to the OWL `property` URI field of the respective attribute map for translation tables of attribute maps (or, if it is empty, the value that would have been generated for it if it weren't empty).

⁷ The OWL individuals described by the OSL document representing the certain types of OBDA maps must be of the OWL types specified in Table 4.2 (for base IRIs, see Paragraph 4 of this specification).

⁸ The OWL properties described by the OSL document representing the fields of the certain OBDA maps must have the IRIs specified in Table 4.3 (for base IRIs, see Paragraph 4 of this specification).

⁹ The following OWL properties in the OSL document refer to lists of elements:

```

osl:rm__sourceColumns
osl:rm__targetColumns
osl:tt__sourceValues
osl:tt__rdfResources

```

Therefore, they have the OWL class `osl:StringListNode` as their range, as is required by the OSL header. They must connect the respective individual to an `osl:StringListNode` individual in every case. This “root node” must *not* have an `osl:hasValue` property.

Map type	Field label	Field name	OWL IRI
Entity map	E1	Table name	osl:em__tableName
Entity map	E2	Label	osl:em__label
Entity map	E3	Identifier map	osl:em__identifierMap
Entity map	E4	Attribute maps...	osl:em__attributeMaps
Entity map	E5	OWL class URI	osl:em__owlClassURI
Entity map	E6	Description	osl:em__description
Attribute map	A1	Column name	osl:am__columnName
Attribute map	A2	SQL datatype	osl:am__sqlDatatype
Attribute map	A3	Mandatory	osl:am__mandatory
Attribute map	A4	Label	osl:am__label
Attribute map	A5	OWL property URI	osl:am__owlPropertyURI
Attribute map	A6	Property type	osl:am__propertyType
Attribute map	A7	Translation	osl:am__translation
Attribute map	A8	URI pattern	osl:am__uriPattern
Attribute map	A9	RDF language	osl:am__rdfLanguage
Attribute map	A10	XSD datatype	osl:am__xsdDatatype
Attribute map	A11	Description	osl:am__description
Identifier map	I1	Entity map	osl:im__entityMap
Identifier map	I2	Attribute maps...	osl:im__attributeMaps
Identifier map	I3	URI pattern	osl:im__uriPattern
Relation map	R1	Source entity map	osl:rm__sourceEntityMap
Relation map	R2	Source column	osl:rm__sourceColumn s
Relation map	R3	Target entity map	osl:rm__targetEntityMap
Relation map	R4	Target column	osl:rm__targetColumn s
Relation map	R5	OWL property URI	osl:rm__owlPropertyURI
Subtype map	S1	Entity Map	osl:sm__entityMap
Subtype map	S2	Column Name	osl:sm__columnName
Subtype map	S3	OWL superclass URI	osl:sm__owlSuperclassURI
Subtype map	S4	Prefix	osl:sm__prefix
Subtype map	S5	Suffix	osl:sm__suffix
Subtype map	S6	Translation	osl:sm__translation
Translation table	T1	Source value...	osl:tt__sourceValue s
Translation table	T2	RDF ressource...	osl:tt__rdfRessource s

Table 4.3.: OWL property IRIs in OSL

If the represented list is not empty, the list elements are represented by other `osl:StringListNode` individuals connected serially by the property `osl:nextNode`, with the first individual being connected to the root node. The node representing the last list element must not have an `osl:nextNode` property.

All nodes except the root node *may* have an `osl:hasValue` property connecting them to their values. The actual list consists of exactly these values, thus, nodes without values are ignored. It is recommended to enumerate the node IRIs, using 0 for the root node.

4.4. An example OSL specification

The following example OSL specification shows how a short OSL document could look like.

4.4.1. The example database schema

Consider, as a simple example database schema a single database table “Persons” containing a primary key column “Id” of type `INTEGER` and a foreign key column “Spouse” referencing the primary key column.

4.4.2. The resulting example OSL specification

The following OSL document will result from the example database schema being bootstrapped with DB2OSL (see Section 5 and Section 6), which uses the XML serialization of OWL. It was shortened by excluding entries representing empty strings, which does not considerably change its semantics while saving some space.

Note the use of a simple URI generation scheme not meeting the requirements described in Section 3.3.1, as is explained in Section 3.1.3. For a more appropriate URI generation scheme introduced in this thesis, refer to Section 3.3.

```
<?xml version="1.0"?>
<rdf:RDF xmlns="http://db2osl_generated_osl_specification/of_spec/test#"
xml:base="http://db2osl_generated_osl_specification/of_spec/test"
xmlns:rdfs="http://www.w3.org/2000/01/rdf-schema#"
xmlns:ont="http://db2osl_generated_osl_specification/of_spec/test#"
xmlns:xsd="http://www.w3.org/2001/XMLSchema#"
xmlns:owl="http://www.w3.org/2002/07/owl#"
xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
xmlns:xml="http://www.w3.org/XML/1998/namespace"
xmlns:osl="http://w3studi.informatik.uni-stuttgart.de/~martispp/ont#">
<owl:Ontology rdf:about="http://db2osl_generated_osl_specification/of_spec/test">
<owl:imports rdf:resource=
    "http://w3studi.informatik.uni-stuttgart.de/~martispp/ont/db2osl.owl"/>
</owl:Ontology>
```

```

<!--
////////////////////////////////////
//
// Individuals
//
////////////////////////////////////
-->

<!-- #test:Persons__ENTITY_MAP -->
<owl:NamedIndividual rdf:about="#test:Persons__ENTITY_MAP">
<rdf:type rdf:resource=
    "http://w3studi.informatik.uni-stuttgart.de/~martispp/ont#EntityMap"/>
<osl:em__attributeMaps rdf:resource=
    "#test:table_Persons__column_Id__ATTRIBUTE_MAP"/>
<osl:em__attributeMaps rdf:resource=
    "#test:table_Persons__column_Spouse__ATTRIBUTE_MAP"/>
<osl:em__identifierMap rdf:resource="#test:Persons__IDENTIFIER_MAP"/>
<osl:em__owlClassURI rdf:datatype="http://www.w3.org/2001/XMLSchema#string">
    test:Persons
</osl:em__owlClassURI>
<osl:em__tableName rdf:datatype="http://www.w3.org/2001/XMLSchema#string">
    Persons
</osl:em__tableName>
</owl:NamedIndividual>

<!-- #test:Persons__IDENTIFIER_MAP -->
<owl:NamedIndividual rdf:about="#test:Persons__IDENTIFIER_MAP">
<rdf:type rdf:resource=
    "http://w3studi.informatik.uni-stuttgart.de/~martispp/ont#IdentifierMap"/>
<osl:im__attributeMaps rdf:resource=
    "#test:table_Persons__column_Id__ATTRIBUTE_MAP"/>
<osl:im__entityMap rdf:resource="#test:Persons__ENTITY_MAP"/>
<osl:im__uriPattern rdf:datatype="http://www.w3.org/2001/XMLSchema#string">
    test:id/{$1}/
</osl:im__uriPattern>
</owl:NamedIndividual>

<!-- #test:fk_Spouse__RELATION_MAP -->
<owl:NamedIndividual rdf:about="#test:fk_Spouse__RELATION_MAP">
<rdf:type rdf:resource=
    "http://w3studi.informatik.uni-stuttgart.de/~martispp/ont#RelationMap"/>
<osl:rm__sourceColumns rdf:resource=
    "#test:fk_Spouse__RELATION_MAP__SOURCE_COLUMNS__NODE_0"/>
<osl:rm__sourceEntityMap rdf:resource="#test:Persons__ENTITY_MAP"/>
<osl:rm__targetColumns rdf:resource=
    "#test:fk_Spouse__RELATION_MAP__TARGET_COLUMNS__NODE_0"/>

```

```

<osl:rm__targetEntityMap rdf:resource="#test:Persons__ENTITY_MAP"/>
<osl:rm__owlPropertyURI rdf:datatype="http://www.w3.org/2001/XMLSchema#string">
    test:fk_Spouse
</osl:rm__owlPropertyURI>
</owl:NamedIndividual>

<!-- #test:fk_Spouse__RELATION_MAP__SOURCE_COLUMNS__NODE_0 -->
<owl:NamedIndividual rdf:about=
    "#test:fk_Spouse__RELATION_MAP__SOURCE_COLUMNS__NODE_0">
<rdf:type rdf:resource=
    "http://w3studi.informatik.uni-stuttgart.de/~martispp/ont#StringListNode"/>
<osl:nextNode rdf:resource=
    "#test:fk_Spouse__RELATION_MAP__SOURCE_COLUMNS__NODE_1"/>
</owl:NamedIndividual>

<!-- #test:fk_Spouse__RELATION_MAP__SOURCE_COLUMNS__NODE_1 -->
<owl:NamedIndividual rdf:about=
    "#test:fk_Spouse__RELATION_MAP__SOURCE_COLUMNS__NODE_1">
<rdf:type rdf:resource=
    "http://w3studi.informatik.uni-stuttgart.de/~martispp/ont#StringListNode"/>
<osl:hasValue rdf:datatype="http://www.w3.org/2001/XMLSchema#string">
    Spouse
</osl:hasValue>
</owl:NamedIndividual>

<!-- #test:fk_Spouse__RELATION_MAP__TARGET_COLUMNS__NODE_0 -->
<owl:NamedIndividual rdf:about=
    "#test:fk_Spouse__RELATION_MAP__TARGET_COLUMNS__NODE_0">
<rdf:type rdf:resource=
    "http://w3studi.informatik.uni-stuttgart.de/~martispp/ont#StringListNode"/>
<osl:nextNode rdf:resource=
    "#test:fk_Spouse__RELATION_MAP__TARGET_COLUMNS__NODE_1"/>
</owl:NamedIndividual>

<!-- #test:fk_Spouse__RELATION_MAP__TARGET_COLUMNS__NODE_1 -->
<owl:NamedIndividual rdf:about=
    "#test:fk_Spouse__RELATION_MAP__TARGET_COLUMNS__NODE_1">
<rdf:type rdf:resource=
    "http://w3studi.informatik.uni-stuttgart.de/~martispp/ont#StringListNode"/>
<osl:hasValue rdf:datatype="http://www.w3.org/2001/XMLSchema#string">
    Id
</osl:hasValue>
</owl:NamedIndividual>

<!-- #test:table_Persons__column_Id__ATTRIBUTE_MAP -->
<owl:NamedIndividual rdf:about="#test:table_Persons__column_Id__ATTRIBUTE_MAP">
<rdf:type rdf:resource=

```

```

    "http://w3studi.informatik.uni-stuttgart.de/~martispp/ont#AttributeMap"/>
<osl:am__columnName rdf:datatype="http://www.w3.org/2001/XMLSchema#string">
    Id
</osl:am__columnName>
<osl:am__mandatory rdf:datatype="http://www.w3.org/2001/XMLSchema#boolean">
    false
</osl:am__mandatory>
<osl:am__owlPropertyURI rdf:datatype="http://www.w3.org/2001/XMLSchema#string">
    test:table_Persons__column_Id
</osl:am__owlPropertyURI>
<osl:am__propertyType rdf:datatype="http://www.w3.org/2001/XMLSchema#string">
    DataProperty
</osl:am__propertyType>
<osl:am__sqlDatatype rdf:datatype="http://www.w3.org/2001/XMLSchema#string">
    INTEGER
</osl:am__sqlDatatype>
</owl:NamedIndividual>

<!-- #test:table_Persons__column_Spouse__ATTRIBUTE_MAP ->
<owl:NamedIndividual rdf:about="#test:table_Persons__column_Spouse__ATTRIBUTE_MAP">
<rdf:type rdf:resource=
    "http://w3studi.informatik.uni-stuttgart.de/~martispp/ont#AttributeMap"/>
<osl:am__columnName rdf:datatype="http://www.w3.org/2001/XMLSchema#string">
    Spouse
</osl:am__columnName>
<osl:am__mandatory rdf:datatype="http://www.w3.org/2001/XMLSchema#boolean">
    false
</osl:am__mandatory>
<osl:am__propertyType rdf:datatype="http://www.w3.org/2001/XMLSchema#string">
    ObjectProperty
</osl:am__propertyType>
<osl:am__sqlDatatype rdf:datatype="http://www.w3.org/2001/XMLSchema#string">
    INTEGER
</osl:am__sqlDatatype>
</owl:NamedIndividual>

<!--
////////////////////////////////////
//
// General axioms
//
////////////////////////////////////
-->
<rdf:Description>
<rdf:type rdf:resource="http://www.w3.org/2002/07/owl#AllDifferent"/>
<owl:distinctMembers rdf:parseType="Collection">
<rdf:Description rdf:about="#test:Persons__ENTITY_MAP"/>

```

```

<rdf:Description rdf:about="#test:Persons__IDENTIFIER_MAP"/>
<rdf:Description rdf:about="#test:fk_Spouse__RELATION_MAP"/>
<rdf:Description rdf:about="#test:fk_Spouse__RELATION_MAP__SOURCE_COLUMNS__NODE_0"/>
<rdf:Description rdf:about="#test:fk_Spouse__RELATION_MAP__SOURCE_COLUMNS__NODE_1"/>
<rdf:Description rdf:about="#test:fk_Spouse__RELATION_MAP__TARGET_COLUMNS__NODE_0"/>
<rdf:Description rdf:about="#test:fk_Spouse__RELATION_MAP__TARGET_COLUMNS__NODE_1"/>
<rdf:Description rdf:about="#test:table_Persons__column_Id__ATTRIBUTE_MAP"/>
<rdf:Description rdf:about="#test:table_Persons__column_Spouse__ATTRIBUTE_MAP"/>
</owl:distinctMembers>
</rdf:Description>
</rdf:RDF>

```

4.5. Conclusion on the OBDA Specification Language (OSL)

The OBDA SPECIFICATION LANGUAGE reaches its objectives pretty well. Especially the basing on OWL brings major advantages, like, of course, the ability to process and edit OSL specifications with common OWL tools. Also, a potential user probably to some degree is familiar with this environment, making it much easier to learn how to deal with OSL documents.

There are, however, some shortcomings from the approach of using OWL as it was realized: the different serialization formats of OWL [W3C12] do not guarantee the data being represented as a text file and thus being human-readable and suitable for the web. Albeit losing some flexibility and the ability to store OSL documents in a compact and performance-advantaging binary format, the allowed serialization formats should have been restricted.

Regarding performance and compactness, of course a binary format would be much more advantageous, but since these were not considered important issues, nor explicit requirements, this was not seriously considered. In any case, it would have been possible to develop a completely own approach meeting all of the requirements. This would have had the advantage of being able to much better achieve the objectives of readability, changeability and simpleness of implementations. However, this also was not seriously considered, since it would have offset many of the advantages described previously, while most probably not being flawless in the first years of its use. Besides, the language specification then would have been much more extensive and complicated. When implementing the DB2OSL software, the specification was perceived compact and concise while leaving no questions open.

Finally, OWL is seen as an ideal base for the OBDA SPECIFICATION LANGUAGE, being a solid framework for data and constraint representation with a high degree of software support, while imposing only a minimum of introductory preparation to the user, and thereby offsetting the disadvantages. Mainly because of this solid foundation, and due to the attempts to represent concepts in an easy and direct way, all requirements were met, apart from the fact that the current definition of the OBDA SPECIFICATION LANGUAGE allows users to use OWL serialization formats which are not human-readable and not suitable for publication in the web. However, it must carefully be considered how to circumvent this shortcoming: by disallowing certain serialization formats, by refining the definition of the OBDA SPECIFICA-

TION LANGUAGE to define several realizations, or by relinquishing from the requirement to have a human-readable format guaranteed.

The specification will definitely be changed in the future, to replace URIs like `http://w3studi.informatik.uni-stuttgart.de/~martispp/ont/db2os1.owl` to more appropriate ones. On this occasion, the previously described shortcoming can be addressed.

5. The db2osl software

Besides the conception of the OBDA SPECIFICATION LANGUAGE (OSL), the design and implementation of the DB2OSL software was an important part of this work. The program itself and its creation process are described in the following sections: Section 5.2 describes the functionality the program offers. Section 5.3 describes how this functionality is exposed to the program environment. Section 5.5 describes the program architecture both on a general and a detailed level. Implementation topics are dealt with in Chapter 6. Section 5.6 finally gives a conclusion about how well the stated requirements on the DB2OSL software were reached. For a list of all classes contained in each package of DB2OSL, refer to Appendix A.1.

The sections of this chapter present the information in a functionally-structured fashion: the concepts and decisions are described along with the topics to which they are linked and the problems that caused them to come about.

5.1. Requirements on db2osl

DB2OSL is required to automatically and correctly bootstrap OBDA specifications from relation database schemata (for details, refer to Section 2.1.3 – OBDA specifications and Section 3.2 – Ontology bootstrapping using OBDA specifications), while not being exploitable by malicious database schemata, nor crashing or exposing undesired behavior in any other occasion.

Also, the database to be bootstrapped must be able to be chosen in a reasonable manner. Of course, the database server to use must be freely selectable, even if it runs on a non-standard port. At least the retrieval of database schemata from databases from an SQL server must be supported. Authentication at an SQL server using a password must be possible. Additionally, all actions of the program must be scriptable, thus being executable from the command-line without user interaction.

Furthermore, the program should be easy to use and provide explanations about how to use it in an unobtrusive, conventional manner. Finally, the code of DB2OSL has to be extremely clean and readable, as it has to be maintained and read by diverse parties.

5.2. Functionality

As described in the Section 5.1 – Requirements on DB2OSL of this thesis, the DB2OSL software is a program automatically deriving an OBDA specification from a relational database schema, which then can be used by other tools to drive the actual bootstrapping process. Its functionality is described in this section, leaving out self-evident features, and is then listed completely.

The way in which this functionality is exposed to users is described in Section 5.3 – [Command line interface](#). The bootstrapping process as the core functionality of the software is described in Section 3.2 – [Ontology bootstrapping using OBDA specifications](#), while the approach of OBDA specifications is described in Section 2.1.3 – [OBDA specifications](#).

The database schema is retrieved by connecting to an SQL database and querying its schema information. Parsing SQL scripts, SQL dumps or other database systems is currently not supported. The databases from which to derive information can be specified by regular expressions, while there are also options to use other databases than the ones specified or even other database servers, taken from a list of hard-coded strings. While these features may not seem to bring about any real benefit at first glance, they proved to be useful for testing purposes. For the same purpose, DB2OSL allows the processing of a hard-coded example database schema.

In addition to OSL output, a low-level output format containing information on all fields of the underlying objects is supported, which is useful for debugging. However, this feature has to be enabled via a minor change in the source code. To allow for customization, the insertion of an own OSL header is supported (for more information on the OSL header, see the specification of the OBDA SPECIFICATION LANGUAGE in Section 4.3). If the standard OSL header is used, it is, by default, loaded from a hard-coded copy; as a result, bootstrapping information from a database server running locally or from the hard-coded example schema requires no Internet connection (simply inserting the `owl:imports` statement of course always works without Internet connection, but the generated underlying ontology is always checked for consistency with the OSL header to prevent the generation of invalid output).

The DB2OSL software can be used both in an interactive and in a non-interactive mode, while skipping a database or a database server or aborting the entire bootstrapping process is possible in either mode. Multiple database servers can be specified for a bootstrapping operation, which then are checked in order for a matching database, allowing to make use of mirrors or fallback servers. Additionally, multiple bootstrapping operations can be specified to be performed in sequence with one invocation of DB2OSL, while all features and settings previously described are enabled, disabled or set per operation. Finally, a help text can be displayed which describes the usage of DB2OSL including the description of all command-line arguments.

The functionality of the DB2OSL software can be summarized as follows:

- Bootstrap one or more OBDA specifications from a database schema by connecting to an SQL database server
- Specify a custom port, login and password for the database server
- Ask for passwords interactively (before starting any bootstrapping operation), hide them if desired
- Specify database names by regular expressions
- Process an arbitrary database if the specified database could not be found
- Connect to a database server containing example databases without having to specify any further details
- Process a hard-coded example database schema without having to specify any further

details

- Use the OSL format described in Chapter 4 – [The OBDA SPECIFICATION LANGUAGE \(OSL\)](#) or a detailed low-level format for output (the latter is for debugging purposes and has to be enabled in the source code)
- Write to standard output or to a file
- Insert a custom OSL header (see the specification of the OBDA SPECIFICATION LANGUAGE (OSL) in Section 4.3 for details)
- Consistency check against a custom OSL header
- Consistency check against the standard OSL header without internet connection
- Act interactively or non-interactively
- Skip currently retrieved database (and try the next one on server), skip current server or abort the overall process at any time, even in non-interactive mode
- Define multiple database servers to search in order for the specified database
- Specify multiple bootstrapping operations to perform in order
- Configure the features described in the above notes per bootstrapping operation
- Display a help text describing the usage of DB2OSL, including the description of all command-line arguments

5.3. Command line interface

This section describes the interface to the operating system and the user interface. For information on programming interfaces, see Section 5.5 – [Architecture of DB2OSL](#).

5.3.1. Command-line arguments

The behavior of DB2OSL itself can be adjusted via command-line arguments (only). Most features can be configured via short options (as, for example, `-P`). To allow for enhanced readability of DB2OSL invocations, each feature can (also) be configured via a long option (like `--password`). The utilization of configuration files was considered, but for the time being, it is seen as unnecessary, for it complicates matters while not addressing any real difficulties.

The command-line arguments DB2OSL currently supports are described in Table 5.1; their default values are listed in Table 5.2. There is currently no switch to set the output format, since the only supported output format, besides OSL, is a low-level output format for debugging purposes. Because of this and since the change that has to be made in the source code to enable it only involves changing one token, it was preferred not to offer a command-line option for this, to not unnecessarily complicate the command-line interface for the normal, non-debugging, user.

The sole invocation of `db2osl`, without any arguments, does not initiate any processing but

Option(s)	Description, taken from the help page of DB2OSL
<code>--database, -d</code>	database name (JAVA regular expression) databases have to match to be processed; see also: <code>--loose-database-match</code>
<code>--echo-password</code>	echo input when prompting for SQL password – must be specified before <code>--password-prompt</code> to become effective
<code>--help, -h,</code>	show this help and exit
<code>--interactive, -i</code>	be interactive when choosing database
<code>--login, -L</code>	SQL login
<code>--loose-database-match</code>	if no database matching the regex specified with <code>--database</code> is found on the given server and <code>--interactive</code> is not specified for this job, use some other database
<code>--osl-header</code>	use the specified custom (non-standard) OSL header, implies <code>--remote-osl-header</code> (to import no header, specify the empty string)
<code>--output-file, -o</code>	use the specified output file (for the standard output, specify “-”)
<code>--password, -P</code>	SQL password; use <code>--password-prompt</code> to get a password prompt (if you do both, the password set via this switch will be ignored)
<code>--password-prompt, -p</code>	prompt for SQL password; a password set via <code>--password</code> is ignored
<code>--remote-osl-header, -R</code>	don’t use hard-coded version of the OSL header for verification
<code>--remote-test</code>	try to retrieve a database schema from a hard-coded list of servers and take the first one successfully retrieved (and accepted, when <code>--interactive</code> is given; note: give a dummy server if you want to do a test besides other jobs)
<code>--test</code>	use hard-coded test database schema, ignore given servers (note: give a dummy server if you want to do a test besides other jobs)

Table 5.1.: Command-line arguments in DB2OSL – descriptions

Option(s)	Default value
<code>--database, -d</code>	<code>.*</code>
<code>--echo-password</code>	<code>false</code>
<code>--help, -h,</code>	<code>false</code>
<code>--interactive, -i</code>	<code>false</code>
<code>--login, -L</code>	<code>anonymous</code>
<code>--loose-database-match</code>	<code>false</code>
<code>--osl-header</code>	<code><empty string></code>
<code>--output-file, -o</code>	<code>-</code>
<code>--password, -P</code>	<code><empty string></code>
<code>--password-prompt, -p</code>	<code>false</code>
<code>--remote-osl-header, -R</code>	<code>false</code>
<code>--remote-test</code>	<code>false</code>
<code>--test</code>	<code>false</code>

Table 5.2.: Command-line arguments in DB2OSL – default values

displays the usage directions instead, in addition to an error message pointing out the missing server argument.

5.3.2. Multiple bootstrapping operations or multiple servers

To perform multiple bootstrapping operations with only one invocation of DB2OSL, it is sufficient to concatenate the command-line arguments for each operation, separated by blanks, to get the final command line. However, when combining a test job with other operations, some arbitrary string has to be inserted as dummy server to enable the program to distinguish the different jobs and to assign each command-line argument to the appropriate job.

Likewise, to check several servers in order for the database to be used for one bootstrapping operation, these servers have to be concatenated, separated by blanks. Again, the distinction of the different bootstrapping jobs has to be possible, so all but the first operation must have at least one command-line argument that signals the beginning of a new job definition (which is not a practical problem, since in order to enforce this, a default argument simply can be stated explicitly without changing the behavior of the invocation).

All settings are configured per operation; as such, when using a shell that separates batched commands by ‘;’,

```
db2osl --database employees --password itsme sql.myemployer.com
--database test myserver.org backup.myserver.org
```

is equivalent to

```
db2osl --database employees --password itsme sql.myemployer.com;
db2osl --database test myserver.org backup.myserver.org
```

Thus, a parameter defined for one operation (like the password in the example) will have no effect on other operations. This ensures that typical errors are prevented when merging several

invocations of DB2OSL into one (or vice versa) and allows additionally for a straight-forward and comprehensible implementation.

5.4. Usage and integration

5.4.1. Basic usage

Currently, the only user interface of DB2OSL is a command-line interface. Since the program is supposed to bootstrap the OBDA specification automatically (and thus there is little interaction, but a lot of output) this was considered ideal. Basically, one invocation of DB2OSL will initiate the automatic, non-interactive bootstrapping of exactly one OSL specification written to the standard output, a behavior which can be modified via command-line arguments. Because of its ability to write to the standard output (which is also the default behavior), it is easy to pipe the output of DB2OSL directly into a program that handles it in a Unix-/POSIX-like fashion [McI87]:

```
db2osl myserver.org | osl2onto myserver.org
```

(supposing OSL2ONTO is a tool that reads an OSL specification from its standard input and uses it to bootstrap an ontology from the database specified on its command line).

This scheme is known as “Pipes and Filters architectural pattern” [BMRSS96].

By inserting additional “filters”, the bootstrapping process can be customized without changing any of the involved programs:

```
db2osl mydatabase.org | customize_spec.sh | osl2onto mydatabase.org
```

(supposing CUSTOMIZE_SPEC.SH is a shell script that modifies a given OSL specification in the way the user desires).

5.4.2. Handling OSL specifications

Since OSL specifications are plain text files, a user can edit them in any desired text editor if he wants to change them in ways that go beyond the functionality DB2OSL provides or that can be achieved by scripts or programs modifying their input automatically. Because of OSL being a subset of OWL (see the specification of OSL in Section 4.3), he can thereby take advantage of editors which support syntax highlighting or other features that make the handling of the respective OWL serialization format more comfortable.

Moreover, every common ontology editor can be used to edit the generated OSL specification automatically or manually. In doing so, care has to be taken to make the ontology remain a conforming OSL specification. However, since the restrictions imposed by OSL are rather small and intuitive, this is easily achieved. One of the most popular ontology editors [MBSF04], PROTÉGÉ, is an open, JAVA based platform supporting plug-ins [NCFK+03]; for a habitual PROTÉGÉ user, it should be an easy task to write an OSL plugin. Furthermore, upcoming tools supporting OSL (see Section 7.2 – Future work) will most likely will be able to check

input files for conformity with the OSL definition.

5.4.3. Integrating db2osl into systems

Besides the use cases described in Paragraph “[Basic usage](#)” and in Paragraph “[Handling OSL specifications](#)” in the previous Section 5.4, there are many other ways in which DB2OSL can be used. For example, a database can be periodically checked for changes that make a re-bootstrapping necessary:

```
db2osl -d mydb myserver.org | sha256sum >oldsum
cp oldsum newsum
while diff oldsum newsum; do # while checksums are the same
    sleep 3600 # wait 1 hour
    db2osl -d mydb myserver.org | sha256sum >newsum
done
rm oldsum newsum
# notify web admin via e-mail:
date | mutt -s "Re-bootstrapping necessary" web-admin@myserver.org
```

Another possible example is the integration of DB2OSL into a shell script that bootstraps all databases on a server:

```
regex='(?!$).*' # accept all nonempty database names first
while db2osl -d "$regex" -o spec myserver.org; do
    dbname="` sed -ne '/xmlns:ont/ { s|.*|/|; s|#||p }' spec `
    mv spec "$dbname".osl
    # don't use this database a second time:
    regex="` printf %s "$regex" | sed -e "s,\\\\\\\\$,,$|dbname$,," `
done
```

To simplify integration on the code level, the architecture of DB2OSL was designed to be highly modular and to cleanly separate code with different areas of responsibility into different packages (for details about the structuring of DB2OSL, see Section 5.5 – [Architecture of DB2OSL](#)). This modularity, besides facilitating comprehension of the code, allows for a high degree of code reusability.

For example, the packages `database`, `osl` and `specification` can be reused in other programs with little or no changes. The biggest change involves combining the database schema retrieval with the user interface of the new program to provide control over the retrieval process when reusing the `database` package (to do this, three method calls have to be replaced). If the accruing information shall be used in a way other than being output or logged, this, of course, must be implemented. If not, it is in many cases sufficient to replace the used `Logger` object by another one providing the desired behavior, since the `Logger` class is part of the JAVA API and widely used [[Gup03](#)]. This is a good example of how using well-known and commonly used classes can greatly improve modularity and reusability.

5.5. Architecture of db2osl

5.5.1. Overall proceeding of db2osl

The overall proceeding of DB2OSL to perform the bootstrapping is roughly outlined in this section. It can be simplistically summarized as illustrated in Figure 5.1. The overall process is a three-step procedure: first, an internal representation is created from the input data, which directly represents database structures like tables, columns, primary keys and foreign keys. Then, the actual bootstrapping is performed on this internal representation, yielding an internal representation of an OBDA specification (for the structure of an OBDA specification, see Section 3.2.1). Finally, an output module is used to output the bootstrapped specification in the desired way.

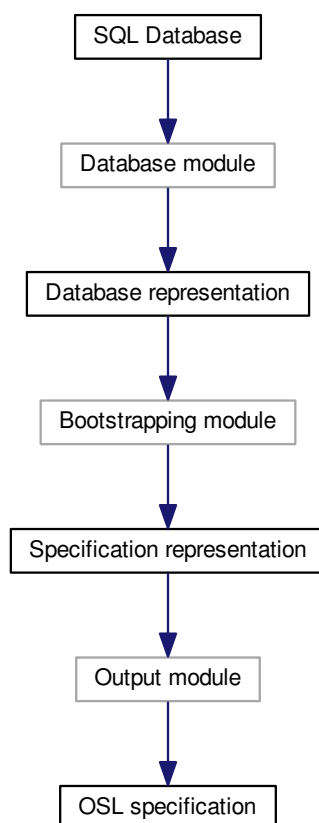


Figure 5.1.: Illustration of the overall proceeding of DB2OSL

This approach decouples the single modules appropriately from each other. Thus, it is easy to add different input or output formats: all that has to be done is to provide transformations from and to the internally used formats, leading to a structure like depicted in Figure 5.2. As this figure illustrates, the DB2OSL software is extendable in an easy and straight-forward manner.

Of course, the interfaces of classes providing the internal representation formats are comprehensible and well-documented. Their description would exceed a scale adequate for this thesis and would be highly repetitive and tedious. Thus, to gain insight into the technical details, re-

fer to the JAVADOC documentation or – for a particularized documentation – to the DOXYGEN documentation (see last paragraphs of Section 6.1 – [Tools employed to create DB2OSL](#)). The packages and class hierarchies and their interaction, however, are described in Section 5.5.3 and Section 5.5.4, respectively.

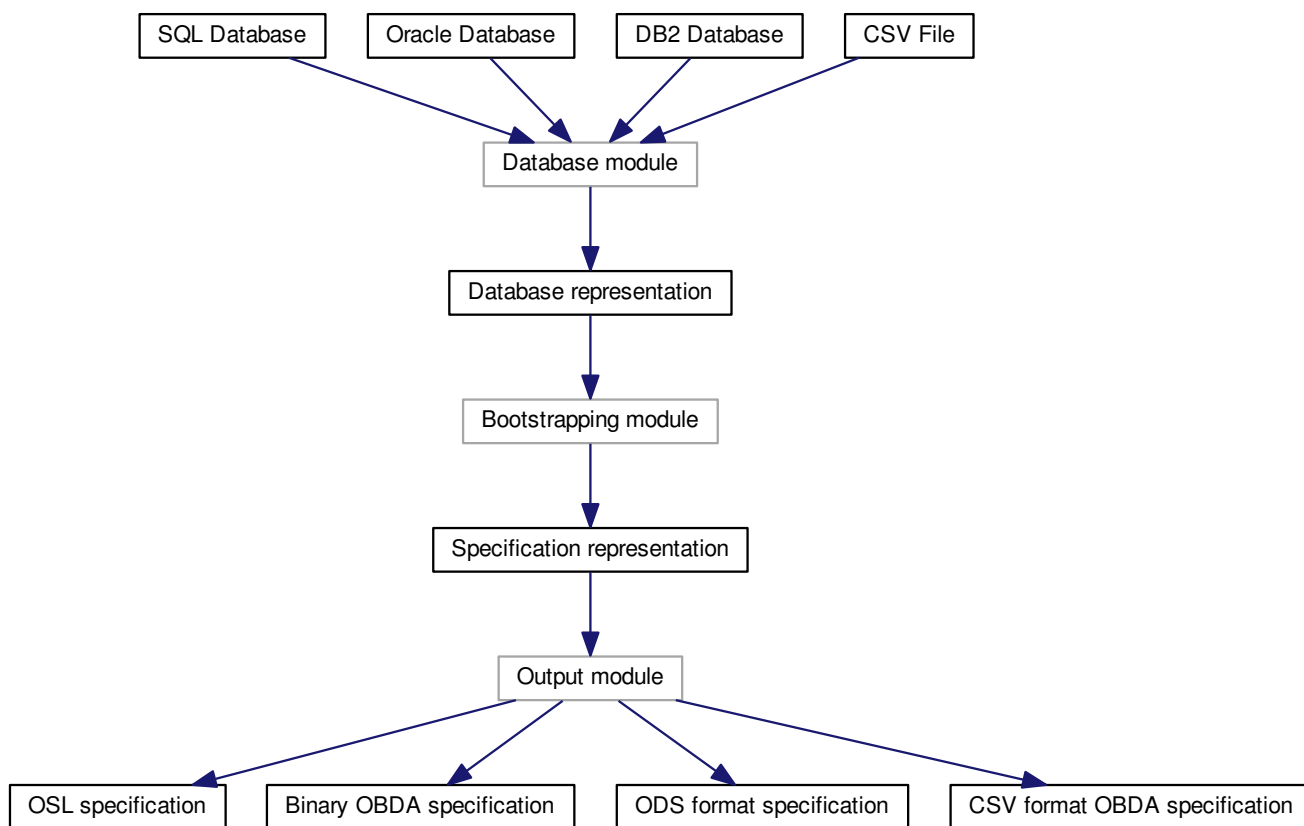


Figure 5.2.: Exemplification of how an extended DB2OSL could proceed

Care was taken to maintain the same modularity in other matters, like logging or the user interface.

5.5.2. Libraries used in db2osl

For database retrieval, the JDBC API was used, which is rather low-level and serves as basis for diverse other libraries, providing meaningful abstractions like database tables, columns, keys and queries. However, none of this libraries was perceived as serving the purpose of providing a simple, well-documented representation of database entities, so this was decided to be carried out by an own implementation.

The OWL API library was used for the representation of OWL ontologies. Note that, even though DB2OSL does not deal with target ontologies, since it only bootstraps the (OBDA) specification of the actual bootstrapping process, it has to represent OWL ontologies due to the basement of its output format OSL on OWL (see Chapter 4 – [The OBDA SPECIFICATION LANGUAGE \(OSL\)](#)). The OWL API was not the first choice, as it was not perceived as well-documented, but it emerged as doing its job of representing OWL ontologies and allowing

to reason on them and validate them pretty well. Note that, although simply named “OWL API”, this library is not part of the JAVA API or in some other way “officially” related to JAVA. It seems, however, to be the primarily recommended and used library for its purpose in the JAVA community. The other alternative considered, JENA, on the other hand, was perceived as being documented very well, but due to its lacking support for OWL 2, its lack of simpleness and imperfect support by the community and other tools, it was considered unsuitable.

To parse the command line, JCOMMANDER was used, a simple yet efficient library, which, given a class with appropriately typed and annotated members, provides means to populate a data structure with command line parameters, to check them for validity and to produce expressive error messages and a help page on request, while offering support for user-defined additional actions.

5.5.3. Coarse structuring of db2osl

As described in Section 5.5.1 – Overall proceeding of DB2OSL, DB2OSL is structured modularly, allowing it to easily extend and modify it, for example by adding support for additional formats or bootstrapping approaches by carrying out changes in the user interface.

The following text introduces the packages that represent these modules and their interaction. For a description on a more fine-grained level, see Section 5.5.4.

Package structuring of db2osl

The 45 classes of DB2OSL were assigned to 11 packages, each containing classes responsible for the same area of operation or taking over similar roles. Care was taken that package division happened senseful, producing meaningful packages with obvious task fields on the one hand, while on the other hand implementing an incisive separation with a notable degree of decoupling. Packages were chosen not to be nested but to be set out rapidly. Since this doesn't have any functional implications [Sch14], but is rather an implementation detail, this is further explained in Section 6.3.5 – Intuitive use of packages.

The packages are introduced and described in Table 5.3. The lists of classes each package contains are given in Table A.1 in Appendix A.1 – Package contents (DB2OSL).

Package	Description
<code>bootstrapping</code>	Classes performing bootstrapping
<code>cli</code>	Classes related to the command line interface of DB2OSL
<code>database</code>	Classes related to the representation of relational databases and attached tasks
<code>helpers</code>	Helper classes used program-wide
<code>log</code>	Classes related to logging and diagnostic output
<code>main</code>	The <code>Main</code> class
<code>osl</code>	Classes representing OBDA specifications (as described in [SGH ⁺ 15]) using the OBDA SPECIFICATION LANGUAGE (OSL)
<code>output</code>	Classes used to output OBDA specifications as described in [SGH ⁺ 15]
<code>settings</code>	Classes related to program and job settings (including command line parsing)
<code>specification</code>	Classes representing (parts of) OBDA specifications (as described in [SGH ⁺ 15]) directly, without involving OSL
<code>test</code>	Classes offering testing facilities

Table 5.3.: Descriptions of the packages in DB2OSL

Besides intuition, as stated, care was involved when partitioning the program into these packages, which included the analysis of the package interaction under a given structure, and the carrying out of changes to make this structure achieve the desired pronounced decoupling with limited and intelligible dependencies.

The `main` package was introduced to make the `Main` class, which carries information needed by other packages – most prominently, the program name –, `importable` from inside these packages. For this, it is required for `Main` not to reside in the `default` package [Sch14].

Decoupling some of the functionality of a package into a new package – which, in a nesting package structure, most probably would have become a sub-package – and thus sacrificing the benefit of having fewer packages also played a role in some cases. Namely, `osl` is a package on its own instead of being part of the `specification` package, the `bootstrapping` classes also form a package on their own instead of belonging to the `specification` package, the classes of the `log` and the `cli` packages were not merged into one package, although logging currently exclusively happens on the command line, and the functionality of the `test` package, though containing only a few lines of code, was separated into its own package.

Even though the package structure would have become quite simpler with these changes applied – 4 out of 11 packages could have been saved this way – the first aim mentioned – meaningfulness and intuitiveness – was taken seriously and the presented partitioning was considered a more natural and comprehensible structuring, emphasizing different roles and thus being a more proper foundation for future extensions of the program. For example, because the `bootstrapping` package is central to the program and takes over an active, processing role and in that is completely different from the classes of the `specification` package which on their part have a *representing* role, it was considered sensible not to merge these two packages. This undergirds the separation of concerns within the program and stresses that the functionality of the `bootstrapping` package should not interweave with that in the `specification` package, making it easier for both to stay independent and further develop into understandable and suitable units.

Package interaction

As mentioned, the structuring of the packages was driven by the aim to gain a notable amount of decoupling. How this reflected in the dependency structure, thus the classes from other packages that the classes of a package depend on, is described in the following. As was also mentioned, the information presented here also acted back on the package partitioning, which changed in consequence.

Dependencies of or on package **helpers** are not considered in the following, since this package precisely was meant to offer services used by many other packages. In fact, all facilities provided by **helpers** could just as well be part of the JAVA API, but unfortunately are not. The current dependency structure, factoring in this restriction, is shown in Figure 5.3 and reveals a conceivably tidy system of dependencies.

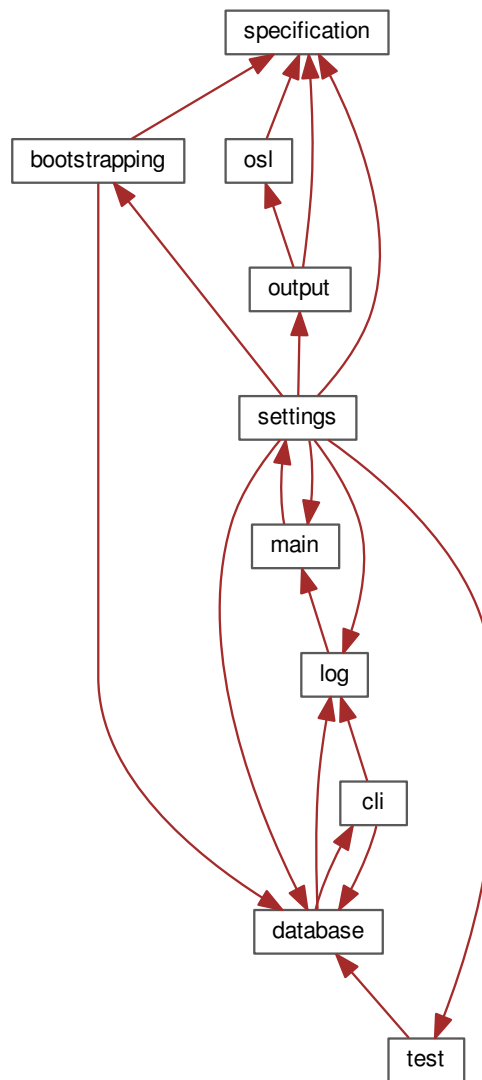


Figure 5.3.: Package dependencies in DB2OSL. “→” means “depends on”.

Except for the package `settings` (which is further explained below), every package has at most two outgoing edges, that is packages it depends on. Previous versions of DB2OSL had a quite more complicated package dependency structure, depicted in Figure 5.4. In this previous package structure, the maximum number of dependencies of packages other than `settings` on other packages is three, which also seems reasonably less. However, in the new structuring, `specification` has no packages it depends on and thus suits its purpose of providing a mundane and straight-forward representation of an OBDA specification much better.

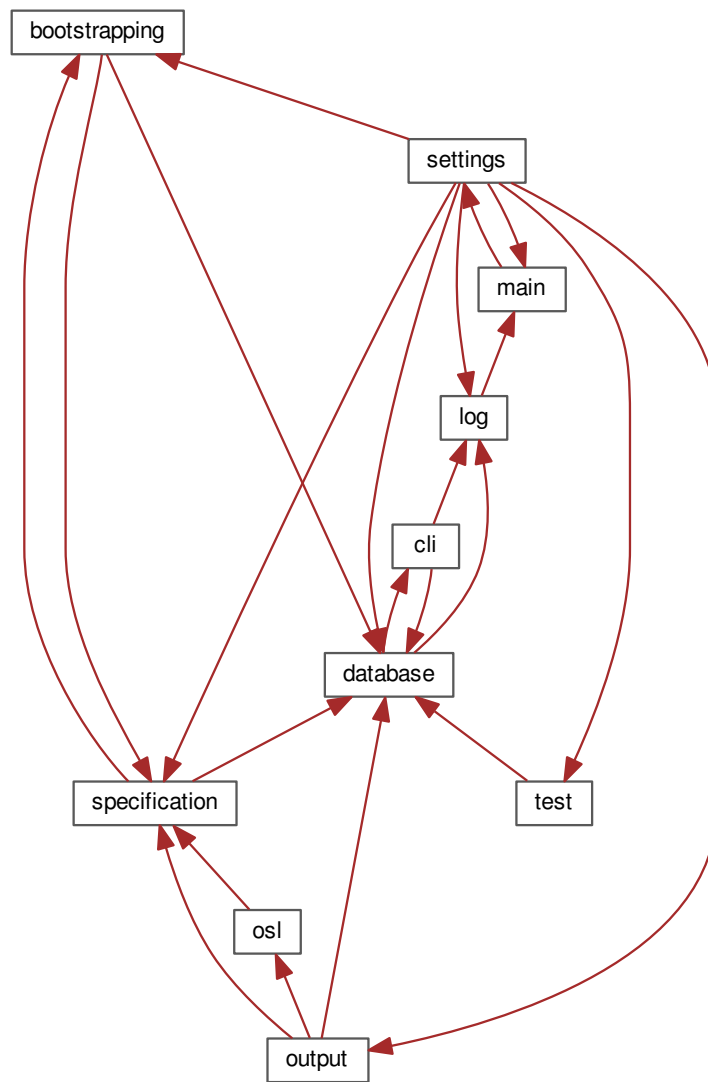


Figure 5.4.: Package dependencies in earlier versions of DB2OSL. “→” again means “depends on”.

Though there still are quite a number of dependencies (to be precise: 19), many of them (8, thus, nearly half) trace back to one central package in the middle, `settings`. This may seem odd at first glance, considering that most of the edges connecting to the `settings` node are outgoing edges and only one is incoming, whereas in a design where the settings are configured from within a single package and accessed from many other packages this would be the other way round. The reason for this constellation is that, as described in Section 5.3 – [Command line interface](#), all settings in DB2OSL are configured per bootstrapping job (there are no global settings) and so `settings` contains a class `Job` (and currently no other classes), which represents the configuration of a bootstrapping job but also provides a `perform()` method combining the facilities offered by the other packages.

By this means, the `perform()` method of the `Job` class acts as the central driver performing

the bootstrapping process, reducing the `main()` method to only 7 lines of code and turning `settings` into something like an externalized part of the `main` package. If, in a future version of the program, this approach is changed and global settings or configuration files are introduced, `settings` will still be the central package, leaving the package structure and dependencies unchanged, since it either way contains information used by many other packages. This was the reason why it was not renamed to, for example, `driver`, which was considered, since at first glance it seems quite a bit unnatural to have the driver class reside in a package called “settings”.

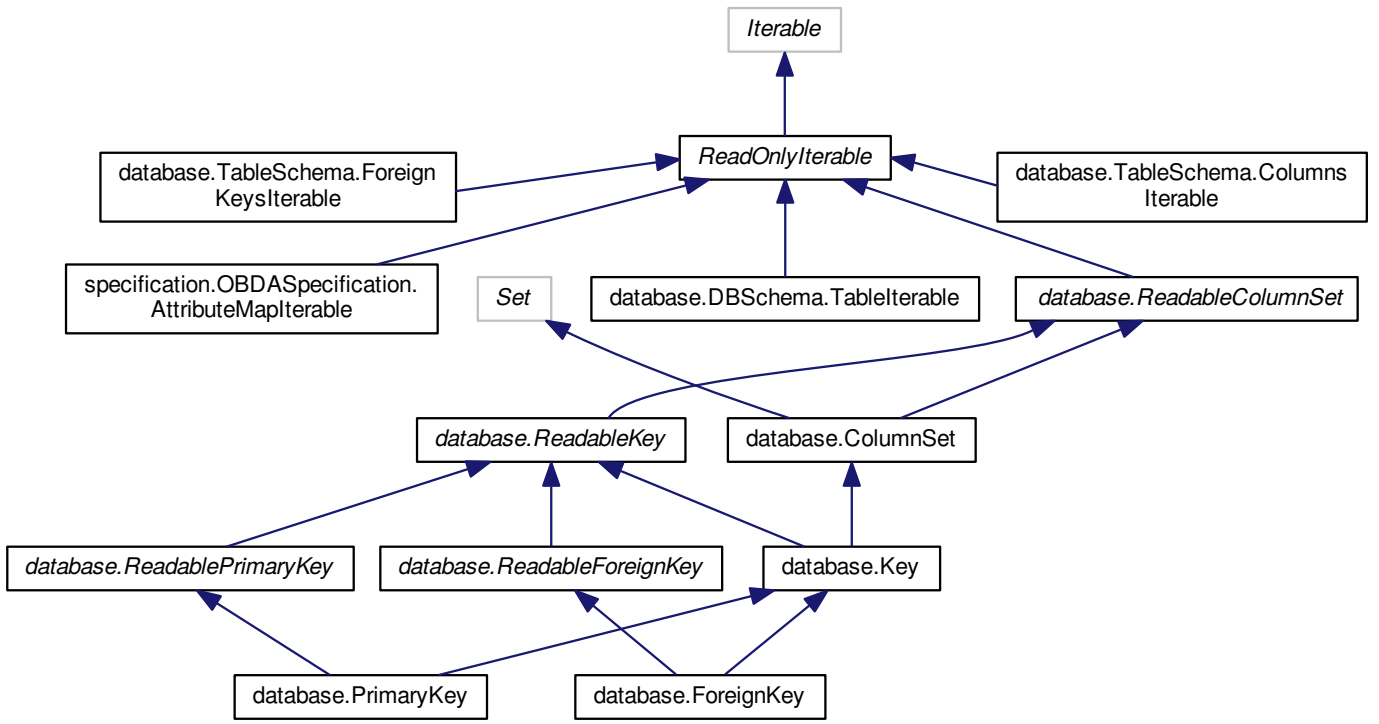
5.5.4. Fine structuring of `db2osl`

While the packages in DB2OSL are introduced and described in Section 5.5.3 – [Coarse structuring of DB2OSL](#), the classes that comprise them are addressed in this section. DB2OSL contains 45 classes in total; for a list of classes contained in each package, refer to Appendix A.1. For information about the program structure on the code level and other implementation issues, see Chapter 6. In particular, Section 6.3.3 – [Intuitive use of classes](#) describes the identification and naming scheme for the classes in DB2OSL. For a detailed class index, subdivided by package affiliation, refer to Appendix A.1.

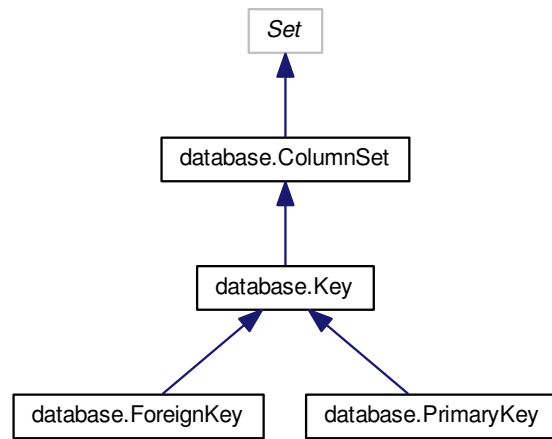
Organizing classes in a structured, obvious manner such that classes have well-defined roles, behave in an intuitive way, ideally representing artifacts from the world modeled in the program directly [Str00], is a prerequisite to make the code clear and comprehensible on the architectural level. This helps to avoid code duplication, to make appropriate use of the type system, to ease the design of precise and flexible interfaces and to enhance the adaptability and extensibility of the program.

The following text describes the class hierarchies and standalone classes of DB2OSL to illustrate how this aims were factored into the design process to ultimately meet the objectives stated in Section 5.1 and to gain insight into the internal structure of the program.

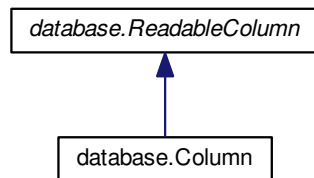
Note that every class hierarchy has at least one `interface` at its top. Classes not belonging to a class hierarchy were chosen not to be given an interface “factitiously”, which would have made them part of a (small) class hierarchy [Sch14]. Deliberately, the scheme often recommended [GHJV95] to give every class an interface it implements was not followed but the approach described by Stroustrup [Str13] to provide a rich set of so called “concrete types” not designed for use within class hierarchies, which “build the foundation of every well-designed program” [Str13]. The details of this consideration are explained in Paragraph “[Avoidance of unnecessary interfaces](#)” in Section 6.3.3 – [Intuitive use of classes](#). In fact, many useful types were already offered by the JAVA API and of course were not re-implemented.



(a) ColumnSet class hierarchy in DB2OSL



(b) ColumnSet class hierarchy in DB2OSL – simplified



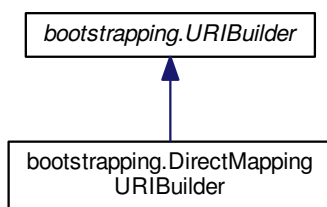
(c) Column class hierarchy in DB2OSL

Figure 5.5.: Database class hierarchies in DB2OSL. Interface names are italicized, external classes or interfaces are hemmed with a gray frame.

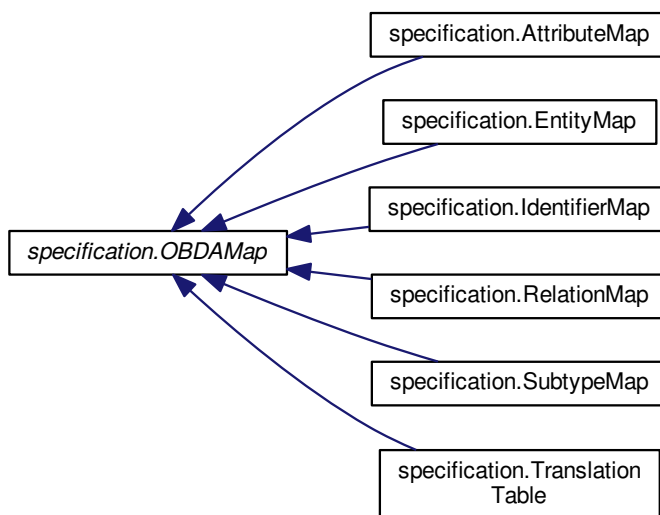
The class hierarchies dealing with databases, depicted in Figure 5.5, definitely include the most complicated class hierarchy of DB2OSL, the class hierarchy around `ColumnSet`, shown in Figure 5.5a. Definitely, it seems overly complicated at the first glance. But this complexity solely is introduced by the artificial `Readable...` interfaces, used to implement const correctness, as described in Section 6.2.2. If JAVA would provide a mechanism like C++'s `const`, this hierarchy would be as simple as in Figure 5.5b.

However, since const correctness is an important mechanism effectively preventing errors while on the other hand introducing clarity by itself, it was considered too important to be sacrificed, even for a cleaner and more intuitive class hierarchy. The fact that the `Readable...` scheme is very straight-forward and a programmer reading the documentation knows about its purpose and the real, much smaller, complexity also makes some amends for the simplicity sacrificed. The const correctness mechanism itself thereby hinders uninformed or ignorant programmers from mistakenly using the wrong class in an interface in many cases.

Another consequence of the approach to implement const correctness is that class `Column` with its interface `ReadableColumn` is an exception of the previously mentioned approach of avoiding factitious interfaces. In fact, it was given an interface although it is basically a concrete type, thus needlessly making it part of a class hierarchy.



(a) URIBuilder class hierarchy in DB2OSL



(b) OBDAMap class hierarchy in DB2OSL

Figure 5.6.: OBDA specification class hierarchies in DB2OSL. Interface names are italicized, external classes or interfaces are hemmed with a gray frame.

Like all other class hierarchies that follow, the class hierarchies dealing with OBDA specifications, depicted in Figure 5.6, are much simpler. In this case, it made sense to represent `URIBuilder` as an interface which is implemented by a concrete class providing URI generation facilities, `DirectMappingURIBuilder`, as shown in Figure 5.6a. This makes the DB2OSL software amenable for future extensions introducing new `URIBuilders` while allowing for flexibility concerning the used `URIBuilder`.

The classes representing the various types of OBDA specification maps depicted in Figure 5.6b were also decided to be given a common interface, although technically this would not have been necessary. In fact, the interface `OBDAMap` does not contain any method but currently only stresses the commonality between these classes. Again, the decision was based on the consideration that possible future extensions might make practical use of this intuitive representation of OBDA specification maps: if, for example, serialization of these types is introduced or the methods are added to their interfaces they have in common, the `OBDAMap` interface is a useful and known extension point for these amendments.

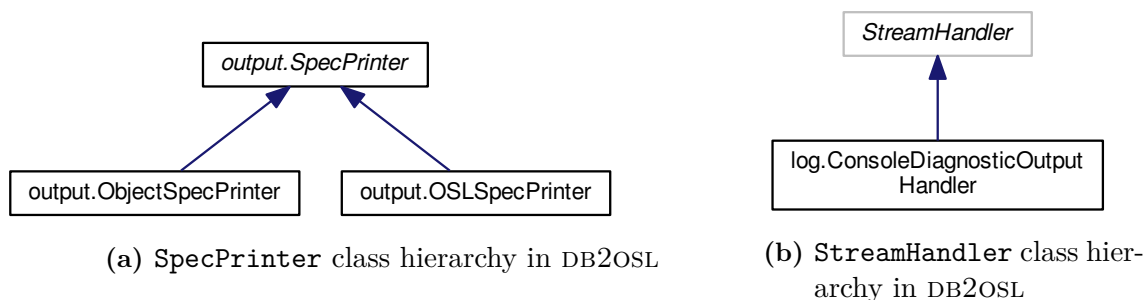


Figure 5.7.: Logging and output class hierarchies in DB2OSL. Interface names are italicized, external classes or interfaces are hemmed with a gray frame.

The hierarchies concerned with logging and output depicted in Figure 5.7 were also kept simple and straight-forward. The hierarchy beyond `SpecPrinter` shown in Figure 5.7a already includes an extension of DB2OSL to support another output format. However, as mentioned in Section 5.2 – **Functionality**, this output format is only meant for debugging purposes and has to be enabled in the source code. The ease of this – one `OSLSpecPrinter` object simply has to be replaced by a `ObjectSpecPrinter` object – demonstrates that the aim of modularity and flexibility is achieved by the used approach.

The hierarchy beyond `StreamHandler` shown in Figure 5.7b, is not a designed hierarchy but simply the implementation of an interface provided by the JAVA API to encapsulate the logging in an own class instead of directly passing `System.err` (which represents the system error output stream in JAVA[Sch14]) to the logger object. However, this can also be seen as a design approach and an extension point.

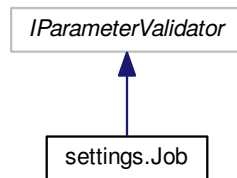


Figure 5.8.: Job class hierarchy in DB2OSL. Interface names are italicized, external classes or interfaces are hemmed with a gray frame.

The class hierarchy around `Job` depicted in Figure 5.8 is simply the mandatory implementation of an interface provided by `JCOMMANDER` (see Section 6.1 – [Tools employed to create DB2OSL](#)) to encapsulate the command-line parsing facilities used by `JCOMMANDER`. Furthermore, the `Job` class acts as the representation of a tuple of settings and as driver class, which is further explained in Section 5.5.3 – [Package interaction](#).

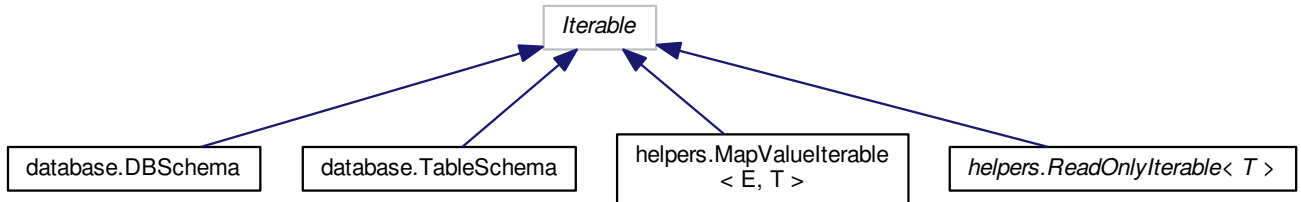
Several miscellaneous class hierarchies are illustrated in Figure 5.9. The hierarchy beyond `Iterable` depicted in Figure 5.9a also is a case of a hierarchy originating from the JAVA API and only included in the DB2OSL software. Furthermore, it exemplifies how the use of common interfaces and concepts facilitates program understanding and implementation: by implementing this interface, all classes sticking to the convention comprehend the implementing classes as representing some kind of set of things. For example, a `DBSchema` represents a set of database tables and a `TableSchema` represents a set of columns. This way, these classes can easily be used in algorithms cleverly designed to rely on this basic interface (only).

The same holds for the hierarchies around `ReadOnlyIterator` and beyond `Iterator`, depicted in Figure 5.9b and Figure 5.9c, respectively: they are solely the inclusion of own classes and interfaces into an existing and widely known concept. The usage of the `Iterator` interface is closely related to the usage of the `Iterable` interface: the latter requires the former in that every class implementing the `Iterable` interface has to be able to return an object of some type implementing the `Iterator` interface [Sch14]. By designing own `Iterator` classes, the traversing of own data structures like an `EntityMap` or `IdentifierMap`, containing several `AttributeMaps` each, or of a `ColumnSet` could be expressed conventionally and easily.

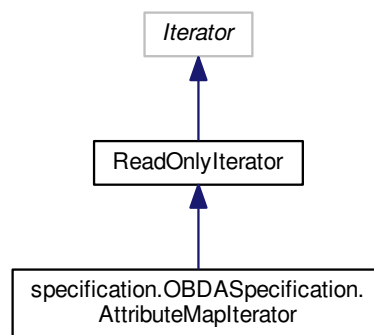
The hierarchies beyond `Exception` and `RuntimeException`, depicted in Figure 5.9d and Figure 5.9e, finally exemplify the same for the concept of exceptions. Designing own exceptions of course was a effective and comprehensible way to deal with errors and other exceptional situations. Again, their introduction would not have been necessary, but was preferred for being more expressive and for more directly representing things than using more abstract exception classes. The `UserAbortException` however, surprisingly could not be taken from the JAVA API which seemingly does not provide an exception class to represent exceptional abortions caused by the user, so it had to be crafted, introducing an additional class hierarchy.

Finally, DB2OSL contains several classes which stand alone and are not part of a hierarchy. They are listed in Table 5.4. All of them are either concrete types or classes only containing static methods, like the `Bootstrapping` or the `Main` class, since it is necessary in Java to introduce a class for these purposes [Sch14]. As notable exception from the otherwise used pattern of carefully designing classes concerning flexibility and future extensions, the `CLIDatabaseInteraction` class primarily is a collection of methods fitting the same purpose and is not general enough, to serve as basis for a systematic extension to provide alternative

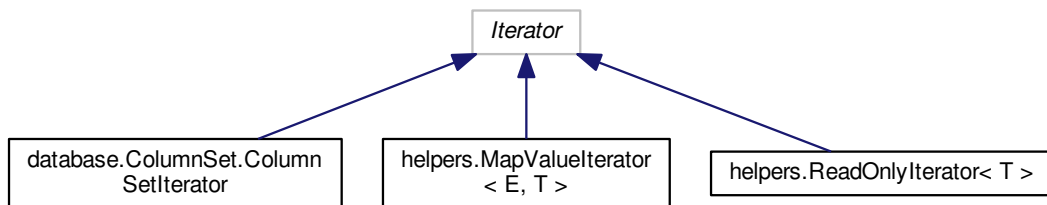
forms of user interaction, though it theoretically can be complemented by another class providing alternative approaches for user interaction in the context of database retrieval. Because of this, it intentionally was not made part of a class hierarchy. Future versions of DB2OSL providing alternative and enhanced forms of user interaction probably should replace it by a well-considered class hierarchy.



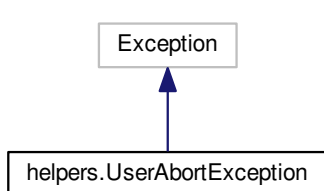
(a) Iterable class hierarchy in DB2OSL



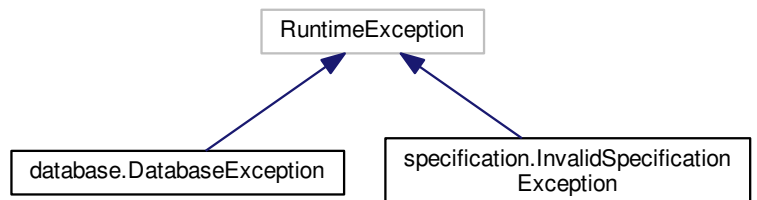
(b) ReadOnlyIterator class hierarchy in DB2OSL



(c) Iterator class hierarchy in DB2OSL



(d) Exception class hierarchy in DB2OSL



(e) RuntimeException class hierarchy in DB2OSL

Figure 5.9.: Miscellaneous class hierarchies in DB2OSL. Interface names are italicized, external classes or interfaces are hemmed with a gray frame.

- `main.Main`
- `database.RetrieveDBSchema`
- `database.Table`
- `helpers.Helpers`
- `helpers.SQLType`
- `specification.OBDA Specification`
- `osl.OSLSpecification`
- `bootstrapping.Bootstrapping`
- `cli.CLIDatabaseInteraction`
- `log.GlobalLogger`
- `test.CreateTestDBSchema`
- `test.GetSomeDBSchema`

Table 5.4.: Standalone classes in DB2OSL

5.6. Conclusion on db2osl

The DB2OSL software achieved the established requirements very well. The requirements regarding processing SQL databases as input source were met, the program is able to work both fully automatic and interactive, while being usable in command-line scripts in a common and intuitive manner. For more details, refer to Section 5.2 – [Functionality](#), to Section 5.3 – [Command line interface](#) and to Section 5.4 – [Usage and integration](#),

In several tests performed by both the author and the Logic and Intelligent Data research group at the University of Oslo, it seemed to work stable and bug-free. Currently, no occasions of crashes or exploits are known concerning the current version.

It furthermore reaches the aim of being amenable for future extensions and being able to be the basis for a library processing OBDA specifications, due to its modularity and the use of well-known concepts and interfaces.

As described more thoroughly in Chapter 6 – [Implementation of DB2OSL](#), the code of DB2OSL was designed to be clean and readable, easing the use and maintenance of DB2OSL by a community of diverse people primary engaged in dealing with concepts and not technical details of the implementation. As also laid out in the previously mentioned section, this aim was reached pretty well. Only some classes of the `database` package, mainly `RetrieveDBSchema` are perceived as being a bit cluttering. However, future extensions probably have their own preferences regarding code style, structuring of classes and packages and the use of comments. When the requirements on these future extensions are clear, the decision if and how this code is restructured is more practicable. Possibly, the basement of the retrieval facilities on parsing facilities is an elegant solution, performing the actual creation of the database representation by a class parsing an SQL script and solely letting a class like `RetrieveDBSchema` fetch the respective SQL script and pass it to the parsing class.

For the same purpose of being usable by diverse users not primarily engaged and interested in implementing, the documentation was written thoughtfully to be complete and correct, releasing users and developers from the need to deal with the source code in many cases. Additionally, a help page is able to be viewed by users without the need to download or deploy

additional files. This was also perceived as useful and advantageous. Thus, the objective an expressive documentation and an unobtrusive, useful help also is seen to be met.

The program can be extended in many ways. If the generation scheme for unique IRIs to be used in the fields of OBDA specification, described in Section 3.3, proves useful, it can be easily integrated into DB2OSL, possibly besides other IRI generation schemes. Furthermore, it is easy to add support for additional input sources or output formats, as illustrated in Section 5.5.1 – [Overall proceeding of DB2OSL](#). The bootstrapping module is also interchangeable and able to be complemented with alternatives.

If the user interface is changed to or complemented with a graphical user interface, the current facilities for this, contained in a single class, `CLIDatabaseInteraction`, should be replaced and a thoughtful design of such an interface introduced. However, for the time being, there is seen no need for this, as the current approach is simple and the requirements on future extensions are not yet known.

6. Implementation of db2osl

This chapter deals with the implementation of the DB2OSL software. Section 6.1 explains what tools were used to create the program. Section 6.2 explains the concepts and decisions that were implemented on the code level to yield robust code. Section 6.3 explains the measures that were taken to yield readable code.

6.1. Tools employed to create db2osl

Several tools were used for the creation of DB2OSL, some of which also proved useful during the creation of this thesis. Their use is described briefly in this section.

Gratitude is offered to the contributors of these tools, all of which are free and open-source software.

Debian GNU/Linux

The operating system on which to run the other tools was DEBIAN GNU/LINUX, version 8.0 (“Jessie”).

Basic Unix tools

Some of the basic Unix shell tools, namely FIND, CAT, GREP, SED, LESS, DIFF and, of course, the shell itself (BASH was used) were very useful. They, for instance, facilitated searching all source code files for common errors or for remains of obsolete constructs that were replaced, for carrying out changes on all source files and for detecting and removing debugging code.

All of the tools were implementations created as part of the GNU project.

git

GIT was used both for version control and for shared access to the source code and related artifacts.

vim

To apply changes involving advanced regular expressions, to perform block editing (insert or remove columns from multiple lines at once), to insert debugging code and similar editing tasks, VIM was very useful.

Eclipse

The IDE to develop the program in was ECLIPSE. It proved very useful particularly due to its abilities to easily create packages and move source files between them as well as to ease the creation of in-code documentation and because of other useful features, like automatic indentation or the automatic insertion of `final` keywords.

OpenJDK

The JAVA compiler, the JAVADOC tool (see next paragraph), the JAVA debugger and the JAVA Runtime Environment used were the implementations provided by the OPENJDK project, version 7.

Javadoc

JAVADOC was used as the primary documentation generation system due to its ability to create clear and well-arranged documentations. Besides, documentations created by JAVADOC are familiar to most JAVA programmers and cleanly integrate into the JAVA environment; for example, methods (automatically) inherited from the `Object` class are incorporated and links to methods and classes provided by the JAVA API are automatically generated.

Doxygen

To complement the documentation generated by JAVADOC (see previous paragraph), DOXYGEN was used, which supports all used JAVADOC constructs. This was particularly useful, because DOXYGEN is able to create a much more in-depth documentation, that includes for instance `private` and `protected` members, the complete source code with syntax highlighting and references to it, and detailed call graphs for all methods. Thus, the DOXYGEN documentation is meant to be a more extensive, detail-oriented documentation providing insight into implementation issues.

6.2. Code robustness

Care was taken to produce code that is geared to incorrect use, making it suitable for the expected environment of sporadic updates by unfamiliar and potentially even unpracticed programmers, who besides have their emphasis on the concepts of bootstrapping rather than details of the present code anyway. In fact, carefully avoiding the introduction of technical artifacts to deal with, preventing programmers from focusing on the actual program logic, is an important principle of writing clean code [Str13] and prevents errors.

In modern object-oriented programming languages, of course the main instruments for achieving this are the type system and exceptions. In particular, static type information should be used to reflect data abstraction and the “kind” of data, an object reflects, while dynamic type information should only be used implicitly, through dynamically dispatching method invocations [Str00]. Exceptions on the other hand should be used at any place related to errors

and error handling, separating error handling noticeably from other code and enforcing the treatment of errors [Str13], preventing the programmer from using corrupted information in many cases.

6.2.1. Enforcing error detection

An example of both mechanisms, static type information and exceptions, acting in combination, while cleanly fitting into the context of dynamic dispatching, are the following methods from `Column.java`:

```
public Boolean isNonNull()  
public Boolean isUnique()
```

Their return type is the JAVA class `Boolean`, not the plain type `boolean`, because the information they return is not always known. In an early stage of the program, they returned `boolean` and were accompanied by two methods `public boolean knownIsNonNull()` and `public boolean knownIsUnique()`, telling the caller whether the respective information was known and thus whether the value returned by `isNonNull()` or `isUnique()`, respectively, was reliable.

They were then changed to return the JAVA class `Boolean` and to return null pointers in case the respective information is not known. This eliminated any possibility of using unreliable data in favor of generating exceptions instead, in this case a `NullPointerException`, which is thrown automatically by the JAVA Runtime Environment [Sch14] if the programmer forgets the null check and tries to get a definite value from one of these methods when the correct value currently is not known.

Comparing two unknown values – thus, two null pointers – also yields the desired result, `true`, since the change, even when the programmer forgets that he deals with objects. However, when comparing two return values of one of the methods in general – as opposed to comparing one such return value against a constant –, errors could occur if the programmer mistakenly writes `col1.isUnique() == col2.isUnique()` instead of `col1.isUnique().booleanValue() == col2.isUnique().booleanValue()`. In this case, since the two `Boolean` objects are compared for identity [Sch14], the former comparison can return `false`, even when the two `boolean` values are in fact the same. However, since this case was considered much less common than cases in which the other solution could make incautious programmers introduce subtle errors, it was preferred. Besides, wrapper classes like `Boolean`, `Integer`, `Long` and `Float` are an integral part of the JAVA language [Sch14], so JAVA programmers were expected to manage to use them properly, so ultimately, since the new solution effectively prevents errors while abstaining from introducing new artifacts, it was considered fair and clean.

6.2.2. Const correctness

Specifying in the code which objects may be altered and which shall remain constant, thus allowing for additional static checks preventing undesired modifications, is commonly referred to as “const correctness” [Str00].

Unfortunately, JAVA lacks a keyword like C++'s `const`, making it harder to achieve const correctness [TE05]. It only specifies the similar keyword `final`, which is much less expressive and doesn't allow for a similarly effective error prevention [TE05]. In particular, because `final` is not part of an object's type information, it is not possible to declare methods that return read-only objects [TE05] – placing a `final` before the method's return type would declare the method `final` [Sch14]. Similarly, there is no way to express that a method must not change the state of its object parameters. A method like `public f(final Object obj)` is only liable to not assigning a new value to its parameter object `obj` [Sch14] (which, if allowed, wouldn't affect the caller anyway [Sch14]). Methods changing its state, on the other hand, are allowed to be called on `obj` without restrictions.

Several possibilities were considered to address this problem:

- Not implementing const correctness, but stating the access rules in comments only
- Not implementing const correctness, but giving the methods which modify object states special names like `setName--USE_WITH_CARE`
- Implementing const correctness by delegating changes of objects to special “editor” objects to be obtained when an object shall be modified
- Implementing const correctness by deriving classes offering the modifying methods from read-only classes

Not implementing const correctness at all of course would have been the simplest possibility, producing the shortest and most readable code, but since incautious manipulation of objects would possibly have introduced subtle, hard-to-spot errors which in many cases would have occurred under additional conditions only and at other places, for example when inserting a `Column` into a `ColumnSet`, this method was not seriously considered.

Not implementing const correctness but using intentionally angular, conspicuous names also was not considered seriously, since it would have cluttered the code for the only sake of hopefully warning programmers of possible errors – and not attempting to avoid them technically.

So the introduction of new classes was considered the most effective and cleanest solution, either in the form of “editor” classes or derived classes offering the modifying methods directly. Again – as during the identification of classes –, the most direct solution was considered the best, so the latter form of introducing additional classes was chosen and classes like `ReadableColumn`, `ReadableColumnSet` et cetera were introduced which offer only the read-only functionality and usually occur in interfaces. Their counterparts including modifying methods also were derived from them and the implications of modifications were explained in their documentation, while the issue and the approach as such were also mentioned in the documentation of the `Readable...` classes. The `Readable...` classes can be converted to their fully-functional counterparts via downcasting (only), thereby giving a strong hint to programmers that the resulting objects are to be used with care.

6.3. Code readability

As the final system hopefully will have a long lifetime cycle and will be used and refined by many people, high code quality was of utmost importance. Beyond architectural issues, this

also involves cleanliness on the lower level, like the design of classes and the implementation of methods. Common software development principles were followed and the unfamiliar reader was constantly taken into account to yield clean, readable and extensible code.

As mentioned in Section 6.3.1 – [Comments](#), an attempt was made to design the code to “speak for itself” as much as possible instead of making its readers depend on comments that provide an understanding. In doing so, other than reducing code size due to the missing comments, clean code amenable to unfamiliar readers and unpredictable changes was enforced. This is especially important since, as described in Section 5.5 – [Architecture of DB2OSL](#), DB2OSL was designed to not only be a standalone program but also to offer components suitable for reusability.

The rest of this section describes these topics in some detail. Besides, an intuitive architecture and suitable, well-designed libraries also contributed to the clarity of the code.

6.3.1. Comments

Comments were used at places where ambiguities or misinterpretations could arise, yet care was taken to face such problems at their roots and solve them wherever possible instead of simply eliminating the ambiguity with comments. This approach rendered many uses of comments unnecessary.

In fact, the number of (plain, e.g. non-JAVADOC) comments was consciously minimized, to enforce speaking code and avoid redundancy. An exception from this was the highlighting of subdivisions. In class and method implementations, comments like

```
//***** Constructors *****\
```

were deliberately used to ease navigation inside source files, but also to enhance readability: parts of method implementations, for example, were optically separated this way. Another alternative would have been to use separate methods for these code pieces, and thereby to stick strictly to the so-called “Composed Method Pattern” [[Bec97](#)], as was done in other cases. However, sticking to this pattern too rigidly would have introduced additional artifacts with either long or non-speaking names, would have interrupted the reading flow and also would have increased complexity, because these methods would have been callable at least from everywhere in the source file. Consequently, having longer methods at some places that are optically separated into smaller units that are in fact independent from each other was considered an elegant solution. Surprisingly though, this technique does not seem to be proposed often in the literature.

Wherever possible, the appropriate JAVADOC comments were used in favor of plain comments, for example, to specify parameters, return types, exceptions and links to other parts of the documentation. This proved even more useful due to the fact that DOXYGEN supports all of the used JAVADOC comments [[Hee16](#)] (but not vice versa [[Ora16](#)]).

6.3.2. Expressive identifier names

Assigning expressive names to methods is a substantially important part of producing speaking code, since methods encapsulate operation and as such are important “building blocks” for other methods [Str13] and, ultimately, for the whole program. Furthermore, method names often occur in interfaces and therefore are not limited to a local scope, nor are they easily changeable without affecting callers [Sch14].

Ultimately, care was taken that method names reflect all important aspects of the respective method’s behavior. Consider the following method from `CLIDatabaseInteraction.java`:

```
public static void promptAbortRetrieveDBSchemaAndWait  
    (final FutureTask<DBSchema> retriever) throws SQLException
```

It could have been called `promptAbortRetrieveDBSchema` only, with the waiting mentioned in a comment. However, the waiting (blocking) is such an important part of its behavior, that this was considered not enough, so the waiting was included in the function name. Since the method is called at one place only, the lengthening of the method name by 7 characters (about 26 %) is not a significant problem.

To keep implementation code readable, care was taken to keep the names of variables meaningful yet concise. If this was not possible, expressiveness was preferred over conciseness.

For example, in the implementation of the database schema retrieval, variables containing data directly obtained from querying the database and thus being subject to further processing was consequently prefixed with “`recvd`”, although in most cases this technically would not have been necessary.

6.3.3. Intuitive use of classes

Following the object-oriented programming paradigm [AO08], classes were heavily used to abstract from implementation details and to yield intuitively usable objects with a set of useful operations.

Meaningful typing includes the direct mapping of entities of the modeled world to code entities [Str13] as well as an expressive naming scheme for the obtained types. Furthermore, inheritance should be used to express commonalities, to avoid code duplication and to separate implementations from interfaces [Str13].

All real-world artifacts to be modeled like database schemata, tables, table schemata, columns, keys and OBDA specifications with their certain map types were directly translated into classes having simple predicting names like `Table`, `TableSchema` and `Key`. Package affiliation provided the correct context to unambiguously understand these names.

To identify potential classes, entities from the problem domain were – if reasonable – directly represented as JAVA classes. The approach of choosing “the program that most directly models the aspects of the real world that we are interested in” to yield clean code, as described and recommended by Stroustrup [Str00], proved to be extremely useful and effective. As a consequence, the code declares classes like `Column`, `ColumnSet`, `ForeignKey`, `Table`, `TableSchema` and `SQLType`. As described in Section 6.3.2 – Expressive identifier names, class names were

chosen to be concise but nevertheless expressive. JAVA packages were used to help attain this aim, which is why the previously mentioned class names are unambiguous. For details about package use, see Section [6.3.5 – Intuitive use of packages](#).

Care was taken not to introduce unnecessary classes, thereby complicating code structure and increasing the number of source files and program entities. Especially artificial classes, having little or no reference to real-world objects, could most often be avoided. On the other hand of course, it usually is not the cleanest solution to avoid such artificial classes entirely.

Section [5.5.4 – Fine structuring of DB2OSL](#) describes how the classes of DB2OSL are organized into class hierarchies.

6.3.4. Avoidance of unnecessary interfaces

In JAVA programming, it is quite common and often recommended [[GHJV95](#)] that every class has at least one `interface` it `implements`, specifying the operations the class provides. If no obvious `interface` exists for a class or the desired interface name is already given to some other entity, the interface is often given names like `ITableSchema` or `TableSchemaInterface`.

However, for a special purpose program with a relatively fixed set of classes mostly representing real-world artifacts from the problem domain, this approach was considered overly cluttering, introducing artificial code entities for no benefit. In particular, as explained in Section [5.5.4 – Fine structuring of DB2OSL](#), all program classes either are standing alone or belong to a class hierarchy derived from at least one interface. So, except from the standalone classes, an interface existed anyway, either “naturally” (as in the case of `Key`, for example) or because of the chosen way to implement `const` correctness. In some cases, these were interfaces declared in the program code, while in some cases, JAVA interfaces like `Set` were implemented (an obvious choice, of course, for `ColumnSet`). Introducing artificial interfaces for the standalone classes was considered unnecessary at least, if not messy.

6.3.5. Intuitive use of packages

As mentioned in Section [6.3.3 – Intuitive use of classes](#), class names were chosen to be concise but nevertheless expressive. This only was possible through the use of JAVA packages, which also helped structure the program.

For the current, relatively limited, extent of the program which currently comprises 45 (`public`) classes, a flat package structure was considered ideal, because it is simple and doesn't stash source files deep in subdirectories (in JAVA, the directory structure of the source tree is required to reflect the package structure [[Sch14](#)]). Because also every class belongs to a package, each source file is to be found exactly one directory below the root program source directory, which in many cases eases their handling.

For the description of the packages, their interaction and considerations on their structuring, see Section [5.5.3 – Coarse structuring of DB2OSL](#). For a detailed package description, refer to Appendix [A.1](#).

Each package is documented in the source code also, namely in a file `package-info.java` residing in the respective package directory. This is a common scheme supported by the ECLIPSE IDE

as well as the documentation generation systems JAVADOC and DOXYGEN (all of which were used in the creation of the program, as described in Section 6.1 – Tools employed to create DB2OSL).

6.3.6. Intuitive control flow

To consequently stick to the maxim of speaking code and further increase readability, an attempt was made to keep control flow intuitive. `do-while` loops, for example, are counter-intuitive: they complicate matters due to the additional, unconditional loop their reader has to keep in mind. Even worse, JAVA's Syntax delays the occurrence of their most important control statement – the loop condition – until after the loop body. Usually, `do-while` loops can be circumvented by properly setting variables influencing the loop condition immediately before the loop and using a `while` loop. Consequently, `do-while` loops were omitted – the code of DB2OSL does not contain a single `do-while` loop.

Another counterproductive technique is the avoidance of the advanced loop control statements `break`, `continue` and `return` and the sole direction of a loop's control flow with its loop condition, often drawing on additional boolean variables like `loopDone` or `loopContinued`. This approach is an essential part of the “structured programming (paradigm)” [Dij72], and its purpose is to enforce that a loop is always left regularly, by unsuccessfully checking the loop condition, which shall ease code verification [Dij72]. A related topic is the general avoidance of the `return` statement (except at the end of a method), for similar considerations [Dij72]. However, both are not needed [Mar08] and, as always, the introduction of artificial technical constructs impairs readability and the ability of the code to “speak for itself”.

Consequently, control flow was not distorted for technical considerations and care was taken to yield straight-forward loops, utilizing advanced control statements to be concise and intuitive and cleverly designed methods that benefit from well-placed `return` statements.

A topic related to intuitive control flow is limited code nesting. Most introduction of new nesting levels greatly increases complexity, since the associated conditions for the respective code to be reached combine with the previous ones in often inscrutable ways. Besides being aware of the execution condition for the code he is currently reading, the reader is forced to either remember the sub-conditions introduced with each nesting level, as well as the current nesting level, or to jump back to the introduction of one or more nestings to figure out the relevant execution condition again.

Naturally, such code is far from being readable and expressive. Thus, overly deep nesting was avoided by rearranging code or using control statements like `return` in favor of opening a new `if` block. The deepest and most complicated nesting in DB2OSL has level 5 (with normal, non-nested method code having level 0), with one of these nestings being dedicated to a big enclosing `while` loop, one to a `try-catch` block and the remaining three to `if` blocks with no `else` parts and trivial one-expression conditions. Additionally, in this case all of the nesting blocks only contained a few lines of code, making the whole construction easily fit on one screen, so this was considered all right. At a few other places there occurs similar, less complicated, nesting up to level 5.

6.3.7. Usage of well-known structures

Great benefit can be taken from constructs familiar to programmers regarding expressiveness. Surely, implementations based on such well-known constructs and patterns are much more likely to be instantly understood by programmers and therefore have a much higher ability of “speaking for themselves”.

Examples in DB2OSL are the (extensively used) iterator concept, const correctness (see Section 6.2.2 – [Const correctness](#)), exceptions, predicates [[Str13](#)], run-time type information [[Str13](#)], helper functions [[Str13](#)] and well-known interfaces from the JAVA API like `Set` or `Collection`, as well as common JAVA constructs, like classes performing a single action (e.g. `OSLSpecPrinter`), and naming schemes, like `get.../set.../is....`

7. Summary and future work

The following sections contain a concluding reflections about the subjects of this work, the objectives and whether they were met and what might be subsequent steps in the future.

7.1. Summary

Unquestionably, the addressing of the difficulties introduced by the complexity of ontology bootstrapping, is a reasonable and gainful issue. The construction of a system centralizing the bootstrapping process within the framework of the OPTIQUE project [KGJR⁺13], while in its part being split into a declarative OBDA specification and a software using this specifications definitely seems worthwhile.

Introducing a well-defined language to represent OBDA specifications surely is an important step towards the goal of establishing an extendable and well-structured system, whose components are loosely coupled and thus interchangeable and flexibly deployable. The objectives of the OBDA SPECIFICATION LANGUAGE are perceived to have been reached, especially the basing on OWL makes the language a useful, comprehensible and easy to integrate tool while allowing for a concise yet complete and expressive specification. Furthermore, it is easy to process and edit OSL specifications with existing tools their users are familiar with and that perfectly fit into the environment of ontologies and the semantic web. Additionally, users will most probably at least to some degree be familiar with OWL itself and thus will easily learn to understand and manually modify OSL specifications. Finally, using an appropriate OWL serialization format, OSL documents are guaranteed to be human-readable text files, thus easily processable and changeable by common tools and ready for exposure in the web.

It should, however, be noticed and maybe perceived as an issue to address that some serialization formats of OWL do not guarantee that representation as a text file [W3C12] and thus the previously mentioned advantages are also not guaranteed.

Regarding DB2OSL, the specified goals were also reached nearly completely. It appears to be a stable software not exposing any undesired behavior, while accomplishing the stated goal of processing SQL databases to retrieve database schemata and generate OBDA specifications in the OSL format from them. As required, the program works fully automatic and is easy to use in scripts, while the intended behavior is easy to read from the command-line expression invoking it. Besides, it offers an interactive mode while in any case allowing to aborting schema retrieval, skip a database server or quit the program at any time. Both the invocation and the integration happen in a commonly used manner and the behavior is intuitive for users familiar with the “Pipes and filters architectural pattern” used in Unix [McI87].

Furthermore, DB2OSL is ready for future extensions and amends and the transformation into a library. Carefully designed to contain clean and readable code, it can be easily used and

maintained by a big and varying community of people not interested in being hindered by technical issues.

The proposed IRI generation scheme also can be useful part of the bootstrapping process. Since proven to be correct, while also being expressive and simple, it fits in the bootstrapping concepts very well.

7.2. Future work

For the future, as mentioned previously, the refinement of the specification of OSL should be considered, regarding the restriction of the allowed OWL serializations. Though losing some possibilities, like compact and performance-advantaging storage of OSL documents, the advantages of guaranteeing human-readability and easy modification might compensate for this. The specification also has to be modified to change URIs like `http://w3studi.informatik.uni-stuttgart.de/~martispp/ont/db2osl.owl`, to more appropriate ones.

Although, of course software processing and validating OSL documents are an important future task.

DB2OSL, finally, should be extended to support more input and output formats and maybe to allow for more customization. It should, however, not move away from its intuitive usability as a command-line tool.

Appendices

A. Details on the db2osl implementation

A.1. Package contents (db2osl)

The following table lists the contents of each of DB2OSLs packages:

- bootstrapping
 - Bootstrapping
 - DirectMappingURIBuilder
 - URIBuilder
- cli
 - CLIDatabaseInteraction
- database
 - Column
 - ColumnSet
 - DatabaseException
 - DBSchema
 - ForeignKey
 - Key
 - PrimaryKey
 - ReadableColumn
 - ReadableColumnSet
 - ReadableForeignKey
 - ReadableKey
 - ReadablePrimaryKey
 - RetrieveDBSchema
 - Table
 - TableSchema
- helpers
 - Helpers
 - MapValueIterable
 - MapValueIterator
 - ReadOnlyIterable
 - ReadOnlyIterator
 - SQLType
 - UserAbortException
- log
 - ConsoleDiagnosticOutputHandler
 - GlobalLogger
- main
 - Main
- osl
 - OSLSpecification
- output
 - ObjectSpecPrinter
 - OSLSpecPrinter
 - SpecPrinter
- settings
 - Job
- specification
 - AttributeMap
 - EntityMap
 - IdentifierMap
 - InvalidSpecificationException
 - OBDAMap
 - OBDASpecification
 - RelationMap
 - SubtypeMap
 - TranslationTable
- test
 - CreateTestDBSchema
 - GetSomeDBSchema

Table A.1.: Class attachment to packages in DB2OSL

B. Details on the OBDA Specification Language (OSL)

B.1. The OSL header

The following pages contain the standard OSL header in the XML serialization of OWL.

```
<?xml version="1.0"?>
<rdf:RDF xmlns="http://w3studi.informatik.uni-stuttgart.de/~martispp/ont#"
xml:base="http://w3studi.informatik.uni-stuttgart.de/~martispp/ont"
xmlns:rdfs="http://www.w3.org/2000/01/rdf-schema#"
xmlns:xsd="http://www.w3.org/2001/XMLSchema#"
xmlns:owl="http://www.w3.org/2002/07/owl#"
xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
xmlns:xml="http://www.w3.org/XML/1998/namespace"
xmlns:osl="http://w3studi.informatik.uni-stuttgart.de/~martispp/ont#">
<owl:Ontology rdf:about=
    "http://w3studi.informatik.uni-stuttgart.de/~martispp/ont/db2osl.owl"/>

<!--
////////////////////////////////////
//
// Object Properties
//
////////////////////////////////////
-->
<!-- #am_translation ->
<owl:ObjectProperty rdf:about="#am_translation">
<rdf:type rdf:resource="http://www.w3.org/2002/07/owl#FunctionalProperty"/>
<rdfs:domain rdf:resource="#AttributeMap"/>
<rdfs:range rdf:resource="#TranslationTable"/>
</owl:ObjectProperty>

<!-- #em_attributeMaps ->
<owl:ObjectProperty rdf:about="#em_attributeMaps">
<rdfs:domain rdf:resource="#EntityMap"/>
<rdfs:range rdf:resource="#AttributeMap"/>
</owl:ObjectProperty>
```

```

<!-- #em__identifierMap ->
<owl:ObjectProperty rdf:about="#em__identifierMap">
<owl:inverseOf rdf:resource="#im__entityMap"/>
<rdf:type rdf:resource="http://www.w3.org/2002/07/owl#FunctionalProperty"/>
<rdf:type rdf:resource="http://www.w3.org/2002/07/owl#InverseFunctionalProperty"/>
<rdfs:domain rdf:resource="#EntityMap"/>
<rdfs:range rdf:resource="#IdentifierMap"/>
</owl:ObjectProperty>

<!-- #im__attributeMaps ->
<owl:ObjectProperty rdf:about="#im__attributeMaps">
<rdfs:domain rdf:resource="#IdentifierMap"/>
<rdfs:range rdf:resource="#AttributeMap"/>
</owl:ObjectProperty>

<!-- #im__entityMap ->
<owl:ObjectProperty rdf:about="#im__entityMap">
<rdf:type rdf:resource="http://www.w3.org/2002/07/owl#FunctionalProperty"/>
<rdf:type rdf:resource="http://www.w3.org/2002/07/owl#InverseFunctionalProperty"/>
<rdfs:domain rdf:resource="#IdentifierMap"/>
<rdfs:range rdf:resource="#EntityMap"/>
</owl:ObjectProperty>

<!-- #nextNode ->
<owl:ObjectProperty rdf:about="#nextNode">
<rdf:type rdf:resource="http://www.w3.org/2002/07/owl#FunctionalProperty"/>
<rdf:type rdf:resource="http://www.w3.org/2002/07/owl#AsymmetricProperty"/>
<rdfs:domain rdf:resource="#StringListNode"/>
<rdfs:range rdf:resource="#StringListNode"/>
</owl:ObjectProperty>

<!-- #rm__sourceColumns ->
<owl:ObjectProperty rdf:about="#rm__sourceColumns">
<rdf:type rdf:resource="http://www.w3.org/2002/07/owl#FunctionalProperty"/>
<rdfs:domain rdf:resource="#RelationMap"/>
<rdfs:range rdf:resource="#StringListNode"/>
</owl:ObjectProperty>

<!-- #rm__sourceEntityMap ->
<owl:ObjectProperty rdf:about="#rm__sourceEntityMap">
<rdf:type rdf:resource="http://www.w3.org/2002/07/owl#FunctionalProperty"/>
<rdfs:domain rdf:resource="#RelationMap"/>
<rdfs:range rdf:resource="#EntityMap"/>
</owl:ObjectProperty>

<!-- #rm__targetColumns ->

```

```

<owl:ObjectProperty rdf:about="#rm__targetColumns">
<rdf:type rdf:resource="http://www.w3.org/2002/07/owl#FunctionalProperty"/>
<rdfs:domain rdf:resource="#RelationMap"/>
<rdfs:range rdf:resource="#StringListNode"/>
</owl:ObjectProperty>

<!-- #rm__targetEntityMap ->
<owl:ObjectProperty rdf:about="#rm__targetEntityMap">
<rdf:type rdf:resource="http://www.w3.org/2002/07/owl#FunctionalProperty"/>
<rdfs:domain rdf:resource="#RelationMap"/>
<rdfs:range rdf:resource="#EntityMap"/>
</owl:ObjectProperty>

<!-- #sm__entityMap ->
<owl:ObjectProperty rdf:about="#sm__entityMap">
<rdf:type rdf:resource="http://www.w3.org/2002/07/owl#FunctionalProperty"/>
<rdfs:domain rdf:resource="#SubtypeMap"/>
<rdfs:range rdf:resource="#EntityMap"/>
</owl:ObjectProperty>

<!-- #sm__translation ->
<owl:ObjectProperty rdf:about="#sm__translation">
<rdf:type rdf:resource="http://www.w3.org/2002/07/owl#FunctionalProperty"/>
<rdfs:domain rdf:resource="#SubtypeMap"/>
<rdfs:range rdf:resource="#TranslationTable"/>
</owl:ObjectProperty>

<!-- #tt__rdfResources ->
<owl:ObjectProperty rdf:about="#tt__rdfResources">
<rdfs:domain rdf:resource="#TranslationTable"/>
<rdfs:range rdf:resource="#StringListNode"/>
</owl:ObjectProperty>

<!-- #tt__sourceValues ->
<owl:ObjectProperty rdf:about="#tt__sourceValues">
<rdfs:domain rdf:resource="#TranslationTable"/>
<rdfs:range rdf:resource="#StringListNode"/>
</owl:ObjectProperty>

<!--
////////////////////////////////////
//
// Data properties
//
////////////////////////////////////
-->
<!-- #am__columnName ->

```

```

<owl:DatatypeProperty rdf:about="#am__columnName">
<rdf:type rdf:resource="http://www.w3.org/2002/07/owl#FunctionalProperty"/>
<rdfs:domain rdf:resource="#AttributeMap"/>
<rdfs:range rdf:resource="http://www.w3.org/2001/XMLSchema#string"/>
</owl:DatatypeProperty>

<!-- #am__description -->
<owl:DatatypeProperty rdf:about="#am__description">
<rdf:type rdf:resource="http://www.w3.org/2002/07/owl#FunctionalProperty"/>
<rdfs:domain rdf:resource="#AttributeMap"/>
<rdfs:range rdf:resource="http://www.w3.org/2001/XMLSchema#string"/>
</owl:DatatypeProperty>

<!-- #am__label -->
<owl:DatatypeProperty rdf:about="#am__label">
<rdf:type rdf:resource="http://www.w3.org/2002/07/owl#FunctionalProperty"/>
<rdfs:domain rdf:resource="#AttributeMap"/>
<rdfs:range rdf:resource="http://www.w3.org/2001/XMLSchema#string"/>
</owl:DatatypeProperty>

<!-- #am__mandatory -->
<owl:DatatypeProperty rdf:about="#am__mandatory">
<rdf:type rdf:resource="http://www.w3.org/2002/07/owl#FunctionalProperty"/>
<rdfs:domain rdf:resource="#AttributeMap"/>
<rdfs:range rdf:resource="http://www.w3.org/2001/XMLSchema#boolean"/>
</owl:DatatypeProperty>

<!-- #am__owlPropertyURI -->
<owl:DatatypeProperty rdf:about="#am__owlPropertyURI">
<rdf:type rdf:resource="http://www.w3.org/2002/07/owl#FunctionalProperty"/>
<rdfs:domain rdf:resource="#AttributeMap"/>
<rdfs:range rdf:resource="http://www.w3.org/2001/XMLSchema#string"/>
</owl:DatatypeProperty>

<!-- #am__propertyType -->
<owl:DatatypeProperty rdf:about="#am__propertyType">
<rdf:type rdf:resource="http://www.w3.org/2002/07/owl#FunctionalProperty"/>
<rdfs:domain rdf:resource="#AttributeMap"/>
<rdfs:range rdf:resource="http://www.w3.org/2001/XMLSchema#string"/>
</owl:DatatypeProperty>

<!-- #am__rdfLanguage -->
<owl:DatatypeProperty rdf:about="#am__rdfLanguage">
<rdf:type rdf:resource="http://www.w3.org/2002/07/owl#FunctionalProperty"/>
<rdfs:domain rdf:resource="#AttributeMap"/>
<rdfs:range rdf:resource="http://www.w3.org/2001/XMLSchema#string"/>
</owl:DatatypeProperty>

```

```

<!-- #am__sqlDatatype -->
<owl:DatatypeProperty rdf:about="#am__sqlDatatype">
<rdf:type rdf:resource="http://www.w3.org/2002/07/owl#FunctionalProperty"/>
<rdfs:domain rdf:resource="#AttributeMap"/>
<rdfs:range rdf:resource="http://www.w3.org/2001/XMLSchema#string"/>
</owl:DatatypeProperty>

<!-- #am__uriPattern -->
<owl:DatatypeProperty rdf:about="#am__uriPattern">
<rdf:type rdf:resource="http://www.w3.org/2002/07/owl#FunctionalProperty"/>
<rdfs:domain rdf:resource="#AttributeMap"/>
<rdfs:range rdf:resource="http://www.w3.org/2001/XMLSchema#string"/>
</owl:DatatypeProperty>

<!-- #am__xsdDatatype -->
<owl:DatatypeProperty rdf:about="#am__xsdDatatype">
<rdf:type rdf:resource="http://www.w3.org/2002/07/owl#FunctionalProperty"/>
<rdfs:domain rdf:resource="#AttributeMap"/>
<rdfs:range rdf:resource="http://www.w3.org/2001/XMLSchema#string"/>
</owl:DatatypeProperty>

<!-- #em__description -->
<owl:DatatypeProperty rdf:about="#em__description">
<rdf:type rdf:resource="http://www.w3.org/2002/07/owl#FunctionalProperty"/>
<rdfs:domain rdf:resource="#EntityMap"/>
<rdfs:range rdf:resource="http://www.w3.org/2001/XMLSchema#string"/>
</owl:DatatypeProperty>

<!-- #em__label -->
<owl:DatatypeProperty rdf:about="#em__label">
<rdf:type rdf:resource="http://www.w3.org/2002/07/owl#FunctionalProperty"/>
<rdfs:domain rdf:resource="#EntityMap"/>
<rdfs:range rdf:resource="http://www.w3.org/2001/XMLSchema#string"/>
</owl:DatatypeProperty>

<!-- #em__owlClassURI -->
<owl:DatatypeProperty rdf:about="#em__owlClassURI">
<rdf:type rdf:resource="http://www.w3.org/2002/07/owl#FunctionalProperty"/>
<rdfs:domain rdf:resource="#EntityMap"/>
<rdfs:range rdf:resource="http://www.w3.org/2001/XMLSchema#string"/>
</owl:DatatypeProperty>

<!-- #em__tableName -->
<owl:DatatypeProperty rdf:about="#em__tableName">
<rdf:type rdf:resource="http://www.w3.org/2002/07/owl#FunctionalProperty"/>
<rdfs:domain rdf:resource="#EntityMap"/>

```

```

<rdfs:range rdf:resource="http://www.w3.org/2001/XMLSchema#string"/>
</owl:DatatypeProperty>

<!-- #hasValue -->
<owl:DatatypeProperty rdf:about="#hasValue">
<rdf:type rdf:resource="http://www.w3.org/2002/07/owl#FunctionalProperty"/>
<rdfs:domain rdf:resource="#StringListNode"/>
<rdfs:range rdf:resource="http://www.w3.org/2001/XMLSchema#string"/>
</owl:DatatypeProperty>

<!-- #im__uriPattern -->
<owl:DatatypeProperty rdf:about="#im__uriPattern">
<rdf:type rdf:resource="http://www.w3.org/2002/07/owl#FunctionalProperty"/>
<rdfs:domain rdf:resource="#IdentifierMap"/>
<rdfs:range rdf:resource="http://www.w3.org/2001/XMLSchema#string"/>
</owl:DatatypeProperty>

<!-- #rm__owlPropertyURI -->
<owl:DatatypeProperty rdf:about="#rm__owlPropertyURI">
<rdf:type rdf:resource="http://www.w3.org/2002/07/owl#FunctionalProperty"/>
<rdfs:domain rdf:resource="#RelationMap"/>
<rdfs:range rdf:resource="http://www.w3.org/2001/XMLSchema#string"/>
</owl:DatatypeProperty>

<!-- #sm__columnName -->
<owl:DatatypeProperty rdf:about="#sm__columnName">
<rdf:type rdf:resource="http://www.w3.org/2002/07/owl#FunctionalProperty"/>
<rdfs:domain rdf:resource="#SubtypeMap"/>
<rdfs:range rdf:resource="http://www.w3.org/2001/XMLSchema#string"/>
</owl:DatatypeProperty>

<!-- #sm__owlSuperclassURI -->
<owl:DatatypeProperty rdf:about="#sm__owlSuperclassURI">
<rdf:type rdf:resource="http://www.w3.org/2002/07/owl#FunctionalProperty"/>
<rdfs:domain rdf:resource="#SubtypeMap"/>
<rdfs:range rdf:resource="http://www.w3.org/2001/XMLSchema#string"/>
</owl:DatatypeProperty>

<!-- #sm__prefix -->
<owl:DatatypeProperty rdf:about="#sm__prefix">
<rdf:type rdf:resource="http://www.w3.org/2002/07/owl#FunctionalProperty"/>
<rdfs:domain rdf:resource="#SubtypeMap"/>
<rdfs:range rdf:resource="http://www.w3.org/2001/XMLSchema#string"/>
</owl:DatatypeProperty>

<!-- #sm__suffix -->
<owl:DatatypeProperty rdf:about="#sm__suffix">

```



```

<rdf:type rdf:resource="http://www.w3.org/2002/07/owl#FunctionalProperty"/>
<rdfs:domain rdf:resource="#SubtypeMap"/>
<rdfs:range rdf:resource="http://www.w3.org/2001/XMLSchema#string"/>
</owl:DatatypeProperty>

<!--
////////////////////////////////////
//
// Classes
//
////////////////////////////////////
-->
<!-- #AttributeMap -->
<owl:Class rdf:about="#AttributeMap">
<rdfs:subClassOf rdf:resource="#OBDAMap"/>
</owl:Class>

<!-- #EntityMap -->
<owl:Class rdf:about="#EntityMap">
<rdfs:subClassOf rdf:resource="#OBDAMap"/>
<owl:hasKey rdf:parseType="Collection">
<rdf:Description rdf:about="#em_owlClassURI"/>
</owl:hasKey>
<owl:hasKey rdf:parseType="Collection">
<rdf:Description rdf:about="#em_tableName"/>
</owl:hasKey>
</owl:Class>

<!-- #IdentifierMap -->
<owl:Class rdf:about="#IdentifierMap">
<rdfs:subClassOf rdf:resource="#OBDAMap"/>
</owl:Class>

<!-- #OBDAMap -->
<owl:Class rdf:about="#OBDAMap">
<owl:disjointWith rdf:resource="#StringListNode"/>
</owl:Class>

<!-- #RelationMap -->
<owl:Class rdf:about="#RelationMap">
<rdfs:subClassOf rdf:resource="#OBDAMap"/>
</owl:Class>

<!-- #StringListNode -->
<owl:Class rdf:about="#StringListNode">
<rdfs:subClassOf rdf:resource="http://www.w3.org/2002/07/owl#Thing"/>
</owl:Class>

```

```

<!-- #SubtypeMap -->
<owl:Class rdf:about="#SubtypeMap">
<rdfs:subClassOf rdf:resource="#OBDAMap"/>
</owl:Class>

<!-- #TranslationTable -->
<owl:Class rdf:about="#TranslationTable">
<rdfs:subClassOf rdf:resource="#OBDAMap"/>
</owl:Class>

<!--
////////////////////////////////////
//
// General axioms
//
////////////////////////////////////
-->
<rdf:Description>
<rdf:type rdf:resource="http://www.w3.org/2002/07/owl#AllDisjointClasses"/>
<owl:members rdf:parseType="Collection">
<rdf:Description rdf:about="#AttributeMap"/>
<rdf:Description rdf:about="#EntityMap"/>
<rdf:Description rdf:about="#IdentifierMap"/>
<rdf:Description rdf:about="#RelationMap"/>
<rdf:Description rdf:about="#SubtypeMap"/>
<rdf:Description rdf:about="#TranslationTable"/>
</owl:members>
</rdf:Description><rdf:Description>
<rdf:type rdf:resource="http://www.w3.org/2002/07/owl#AllDisjointProperties"/>
<owl:members rdf:parseType="Collection">
<rdf:Description rdf:about="#am__columnName"/>
<rdf:Description rdf:about="#am__uriPattern"/>
<rdf:Description rdf:about="#em__owlClassURI"/>
<rdf:Description rdf:about="#em__tableName"/>
<rdf:Description rdf:about="#sm__columnName"/>
<rdf:Description rdf:about="#sm__owlSuperclassURI"/>
</owl:members>
</rdf:Description><rdf:Description>
<rdf:type rdf:resource="http://www.w3.org/2002/07/owl#AllDisjointProperties"/>
<owl:members rdf:parseType="Collection">
<rdf:Description rdf:about="#am__propertyType"/>
<rdf:Description rdf:about="#am__sqlDatatype"/>
<rdf:Description rdf:about="#am__uriPattern"/>
</owl:members>
</rdf:Description>
</rdf:RDF>

```

Bibliography

- [AO08] Prince Oghenekaro Asgba and Edward E. Ogheneovo. “A Comparative Analysis of Structured and Object-Oriented Programming Methods”. In: *Journal of Environmental Management* 12.4 (2008), pp. 41–46 (cit. on p. 65).
- [Bec97] Kent Beck. *Smalltalk Best Practice Patterns*. Prentice-Hall, 1997 (cit. on p. 64).
- [BHBL09] Christian Bizer, Tom Heath, and Tim Berners-Lee. “Linked data – the story so far”. In: *Semantic Services, Interoperability and Web Applications: Emerging Concepts* (2009), pp. 205–227 (cit. on p. 8).
- [BL89] Tim Berners-Lee. *Information Management: A Proposal*. Tech. rep. March 1989, May 1990. CERN, 1989. URL: <http://www.w3.org/History/1989/proposal.html> (cit. on p. 1).
- [BLF99] Tim Berners-Lee and Mark Fischetti. *Weaving the Web: The Original Design and Ultimate Destiny of the World Wide Web by Its Inventor*. 1st. Harper San Francisco, 1999. ISBN: 0062515861 (cit. on p. 1).
- [BMRSS96] Frank Buschmann, Regine Meunier, Hans Rohnert, Peter Sommerlad, and Michael Stal. *Pattern-Oriented Software Architecture - Volume 1: A System of Patterns*. Wiley Publishing, 1996. ISBN: 0471958697, 9780471958697 (cit. on p. 43).
- [BRCGP04] Jesús Barrasa Rodríguez, Óscar Corcho, and Asunción Gómez-Pérez. “R₂O, an Extensible and Semantically Based Database-to-ontology Mapping Language”. In: *SWDB’04: 2nd Workshop on Semantic Web and Databases*. Springer-Verlag, 2004, pp. 1069–1070 (cit. on p. 8).
- [CGH⁺13] D. Calvanese et al. “The Optique Project: Towards OBDA Systems for Industry (Short Paper)”. In: *OWL Experiences and Directions Workshop (OWLED)*. 2013 (cit. on pp. 2, 7, 8).
- [Cro08] J. Crompton. *Keynote talk at the W3C Workshop on Sem. Web in Oil & Gas Industry*. 2008. URL: <http://www.w3.org/2008/12/ogws-slides/Crompton.pdf> (cit. on p. 7).
- [Dij72] E. W. Dijkstra. *Structured Programming*. Ed. by O. J. Dahl, E. W. Dijkstra, and C. A. R. Hoare. London, UK, UK: Academic Press Ltd., 1972. ISBN: 0-12-200550-3 (cit. on p. 67).
- [DS04] Martin Dürst and Michel Suignard. *Internationalized resource identifiers (IRIs)*. Tech. rep. 2004 (cit. on p. 7).

- [EK12] Vadim Eisenberg and Yaron Kanza. “D2RQ/update: updating relational data via virtual RDF”. In: *Proceedings of the 21st international conference companion on World Wide Web*. ACM, 2012, pp. 497–498 (cit. on pp. [iii](#), [iv](#)).
- [GHJV95] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns*. Reading, MA: Addison Wesley, 1995 (cit. on pp. [52](#), [66](#)).
- [Gup03] S. Gupta. *Logging in Java with the JDK 1.4 Logging API and Apache log4j*. Apresspod Series. Apress, 2003. ISBN: 9781590590997 (cit. on p. [44](#)).
- [Hee16] Dimitri van Heesch. *Doxygen: Source code documentation generator tool*. <http://www.doxygen.org>. [Accessed: 2016-05-13]. 2016 (cit. on p. [64](#)).
- [HPZC07] Bin He, Mitesh Patel, Zhen Zhang, and Kevin Chen-Chuan Chang. “Accessing the deep web.” In: *Commun. ACM* 50.5 (2007), pp. 94–101. URL: <http://dblp.uni-trier.de/db/journals/cacm/cacm50.html#HePZC07> (cit. on p. [1](#)).
- [KGJR⁺13] Evgeny Kharlamov et al. “Optique 1.0: Semantic Access to Big Data: The Case of Norwegian Petroleum Directorate’s FactPages”. In: *International Semantic Web Conference (Posters & Demos)*. Ed. by Eva Blomqvist and Tudor Groza. Vol. 1035. CEUR Workshop Proceedings. CEUR-WS.org, 2013, pp. 65–68 (cit. on pp. [7](#), [8](#), [69](#)).
- [KL89] Michael Kifer and Georg Lausen. “F-logic: a higher-order language for reasoning about objects, inheritance, and scheme”. In: *ACM SIGMOD Record*. Vol. 18. 2. ACM, 1989, pp. 134–146 (cit. on p. [8](#)).
- [Mar08] Robert C. Martin. *Clean Code: A Handbook of Agile Software Craftsmanship*. 1st ed. Upper Saddle River, NJ, USA: Prentice Hall PTR, 2008. ISBN: 0132350882, 9780132350884 (cit. on p. [67](#)).
- [MBSF04] J. A. Miller, G. T. Baramidze, A. P. Sheth, and P. A. Fishwick. “Investigating ontologies for simulation modeling”. In: *Simulation Symposium, 2004. Proceedings. 37th Annual*. 2004, pp. 55–63. DOI: [10.1109/SIMSYM.2004.1299465](https://doi.org/10.1109/SIMSYM.2004.1299465) (cit. on p. [43](#)).
- [McI87] M. D. McIlroy. *A Research UNIX Reader: Annotated Excerpts from the Programmer’s Manual, 1971-1986*. Tech. rep. CSTR 139. AT&T Bell Laboratories, 1987 (cit. on pp. [43](#), [69](#)).
- [MNS03] Alexander Maedche, Günter Neumann, and Steffen Staab. “Bootstrapping an ontology-based information extraction system”. In: *Intelligent exploration of the web*. Springer, 2003, pp. 345–359 (cit. on pp. [iii](#), [iv](#), [1](#), [8](#), [18](#)).
- [NCFK⁺03] Natalya F Noy, Monica Crubézy, Ray W Ferguson, Holger Knublauch, Samson W Tu, Jennifer Vendetti, Mark A Musen, et al. “Protege-2000: an open-source ontology-development and knowledge-acquisition environment”. In: *AMIA Annu Symp Proc*. Vol. 953. 2003, p. 953 (cit. on p. [43](#)).

- [Ora16] Oracle Corporation. *javadoc - The Java API Documentation Generator*. <http://docs.oracle.com/javase/7/docs/technotes/tools/windows/javadoc.html>. [Accessed: 2016-05-13]. 2016 (cit. on p. 64).
- [SAM11] Juan F. Sequeda, Marcelo Arenas, and Daniel P. Miranker. “A Completely Automatic Direct Mapping of Relational Databases to RDF and OWL”. In: *International Semantic Web Conference (Posters & Demos)*. Citeseer, 2011 (cit. on p. 8).
- [SAM12] Juan F. Sequeda, Marcelo Arenas, and Daniel P. Miranker. “On Directly Mapping Relational Databases to RDF and OWL”. In: *Proceedings of the 21st International Conference on World Wide Web*. Lyon, France: ACM, 2012, pp. 649–658. ISBN: 978-1-4503-1229-5. DOI: [10.1145/2187836.2187924](https://doi.org/10.1145/2187836.2187924) (cit. on p. 8).
- [Sch14] H. Schildt. *Java: The Complete Reference, Ninth Edition*. The Complete Reference. New York, NY, USA: McGraw-Hill Education, 2014. ISBN: 9780071808552 (cit. on pp. 47, 48, 52, 55, 56, 62, 63, 65, 66).
- [SGH⁺15] Martin G. Skjæveland, Martin Giese, Dag Hovland, Espen H. Lian, and Arild Waaler. “Engineering ontology-based access to real-world data sources”. In: *Web Semantics: Science, Services and Agents on the World Wide Web* 33 (2015), pp. 112–140 (cit. on pp. iii, iv, 1, 2, 4–7, 11–14, 21, 23, 28, 48).
- [SHH⁺09] Satya S Sahoo, Wolfgang Halb, Sebastian Hellmann, Kingsley Idehen, Ted Thibodeau Jr, Sören Auer, Juan Sequeda, and Ahmed Ezzat. “A survey of current approaches for mapping of relational databases to RDF”. In: *W3C RDB2RDF Incubator Group Report* (2009) (cit. on p. 8).
- [SL13] Martin G Skjæveland and Espen H Lian. “Benefits of Publishing the Norwegian Petroleum Directorate’s FactPages as Linked Open Data”. In: *Norsk informatikkonferanse (NIK 2013)*. Tapir (2013) (cit. on p. 8).
- [SLH13] Martin G Skjæveland, Espen H Lian, and Ian Horrocks. “Publishing the Norwegian Petroleum Directorate’s FactPages as Semantic Web Data”. In: *The Semantic Web – ISWC 2013*. Springer, 2013, pp. 162–177 (cit. on p. 8).
- [SSV02] Ljiljana Stojanovic, Nenad Stojanovic, and Raphael Volz. “Migrating data-intensive web sites into the semantic web”. In: *Proceedings of the 2002 ACM symposium on Applied computing*. ACM. 2002, pp. 1100–1107 (cit. on p. 8).
- [STCM11] Juan F. Sequeda, Syed Hamid Tirmizi, Oscar Corcho, and Daniel P. Miranker. “Survey of directly mapping SQL databases to the Semantic Web”. In: *The Knowledge Engineering Review* 26.04 (2011), pp. 445–486 (cit. on p. 8).
- [Str00] Bjarne Stroustrup. *The C++ Programming Language*. 3rd. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2000. ISBN: 0201700735 (cit. on pp. 2, 52, 61, 62, 65).

- [Str13] Bjarne Stroustrup. *The C++ Programming Language*. 4th. Boston, MA, USA: Addison-Wesley Professional, 2013. ISBN: 0321563840, 9780321563842 (cit. on pp. 52, 61, 62, 65, 68).
- [TE05] Matthew S. Tschantz and Michael D. Ernst. “Javari: Adding Reference Immutability to Java”. In: *SIGPLAN Not.* 40.10 (Oct. 2005), pp. 211–230. ISSN: 0362-1340. DOI: [10.1145/1103845.1094828](https://doi.org/10.1145/1103845.1094828) (cit. on p. 63).
- [UG+96] Mike Uschold, Michael Gruninger, et al. “Ontologies: Principles, methods and applications”. In: *Knowledge engineering review* 11.2 (1996), pp. 93–136 (cit. on pp. 7, 13).
- [VDL89] Rick F Van Der Lans. *The SQL standard: a complete guide reference*. Prentice Hall International (UK) Ltd., 1989 (cit. on pp. 11, 23, 25).
- [W3C09] W3C XML Core Working Group. *XML Base (Second Edition)*. <https://www.w3.org/TR/xmlbase/>. [Accessed: 2016-04-02]. 2009 (cit. on p. 29).
- [W3C12] W3C OWL Working Group. *OWL 2 Web Ontology Language - Document Overview (Second Edition)*. <https://www.w3.org/TR/owl2-overview/>. [Accessed: 2016-04-02]. 2012 (cit. on pp. 7, 15, 28, 29, 36, 69).
- [W3C14a] W3C RDF Working Group. *RDF 1.1 Concepts and Abstract Syntax*. <https://www.w3.org/TR/2014/REC-rdf11-concepts-20140225/>. [Accessed: 2016-05-13]. 2014 (cit. on p. 7).
- [W3C14b] W3C RDF Working Group. *RDF Schema 1.1*. <https://www.w3.org/TR/rdf-schema/>. [Accessed: 2016-05-23]. 2014 (cit. on pp. 1, 6, 7, 16).
- [W3CR12a] W3C RDB2RDF Working Group. *A Direct Mapping of Relational Data to RDF*. <https://www.w3.org/TR/rdb-direct-mapping/>. [Accessed: 2016-04-06]. 2012 (cit. on pp. 1, 9–11, 21).
- [W3CR12b] W3C RDB2RDF Working Group. *R2RML: RDB to RDF Mapping Language*. <https://www.w3.org/TR/r2rml/>. [Accessed: 2016-05-20]. 2012 (cit. on pp. 7, 10, 15).

Declaration

I hereby declare that the work presented in this thesis is entirely my own and that I did not use any other sources and references than the listed ones. I have marked all direct or indirect statements from other sources contained therein as quotations. Neither this work nor significant parts of it were part of another examination procedure. I have not published this work in whole or in part before. The electronic copy is consistent with all submitted copies.

place, date, signature