

Czech Technical University in Prague
Faculty of Electrical Engineering
Department of Cybernetics

BUILDING ONTOLOGY-BASED INFORMATION SYSTEMS

Doctoral Thesis

Petr Křemen

Prague, February 2012

Ph.D. Programme: Electrical Engineering and Information Technology

Branch of study: Artificial Intelligence and Biocybernetics

Supervisor: Dr. Zdeněk Kouba

Acknowledgement

First of all, I would like to thank my family. Without support of my wife Zuzana as well as energy and joy that my children David and Jakub passed to me, this work could hardly have been completed.

My great thanks belong to my supervisor Zdeněk Kouba for his fruitful advices, as well as his admirable patience. Also I would like to thank all my colleagues in the Knowledge Based and Software Systems Group at the Department of Cybernetics of the Czech Technical University for creating working environment that was a pleasure for me to work in.

Abstract. Recently, information systems have started to use semantic web ontologies, namely expressed by means of OWL (Web Ontology Language), to store and manage domain knowledge. Comparing to relational databases, ontologies are more flexible to use and provide higher expressiveness for modeling complex domain relationships. However, an information system designer lacks adequate methodologies, techniques and tools that would enable him/her to develop and maintain an information system backed by OWL ontologies. This includes flexible, yet rigor and easy-to-use, programmatic access to OWL ontologies including synchronization of the model of an object-oriented language with the ontology, management of the ontology lifecycle as well as efficient and expressive means for ontology querying.

This thesis addresses these problems and proposes a methodology, as well as techniques and software implementations instantiating the methodology that allow ontology designers to overcome the above described problems. JOPA system, a Java implementation of the techniques introduced in this thesis, offers a semi-automated way for synchronizing an ontology with the application domain model using OWL integrity constraints, as well as efficient and highly expressive query language SPARQL-DL^{NOT} that allows to formulate complex OWL queries. The SPARQL-DL^{NOT} query engine, together with its various optimizations, has been designed as a part of this work and is discussed in detail. To show the feasibility of the proposed methodology, its application on the design of StruFail, a web-based information system for management knowledge about structural failures is described.

Abstrakt. Pro zachycení doménových znalostí se v informačních systémech začínají využívat ontologie sémantického webu, typicky vyjádřené v jazyce OWL (Web Ontology Language). Ve srovnání s relačními databázemi jsou ontologie flexibilnější a poskytují pokročilé vyjadřovací prostředky pro zachycení složitých vztahů v dané expertní oblasti. Návrháři informačních systémů však postrádají vhodné metodiky, techniky a nástroje, které by jim umožnily vyvinout a udržovat informační systémy založené na ontologiích vyjádřených v jazyce OWL. Mezi tyto chybějící metodiky patří formální a současně snadno použitelný způsob programového přístupu k OWL ontologiím zahrnující synchronizaci modelu v daném objektově-orientovaného jazyku s ontologií, správu životního cyklu ontologie a v neposlední řadě rovněž expresivní a efektivní prostředky pro pokládání ontologických dotazů.

Tato disertační práce se věnuje zmíněným problémům a představuje novou metodiku společně se souvisejícími technikami a softwarovými implementacemi, které umožňují návrháři informačního systému výše uvedené problémy překonat. Systém JOPA je realizací představené metodiky v jazyce Java a nabízí možnost semi-automatické synchronizace ontologie s aplikačním doménovým modelem s použitím integritních omezení pro jazyk OWL a rovněž formulovat expresivní dotazy do OWL ontologií pomocí jazyku SPARQL-DL^{NOT}. V této práci je rovněž představen návrh dotazovacího inferenčního stroje v systémech Pellet a OWL2Query včetně několika optimalizačních technik. Závěrem je diskutováno využití představené metodiky a souvisejících technik při návrhu informačního systému StruFail.

Brief Contents

1. Introduction	1
2. Description Logic Reasoning in Semantic Web	7
3. Ontologies in Information Systems	45
4. Proposed Methodology and Framework	55
5. Software Implementation	77
6. Use Cases	91
7. Conclusions	101
Bibliography	102
Acronyms	115
A. SPARQL-DL Atom Abbreviations	117
B. SPARQL-DL ^{NOT} Atom Cost Estimates	119
C. Comparing Ontologies using OWLDiff	121

Contents

1. Introduction	1
1.1. Ontologies in Information Systems	1
1.2. Thesis Contributions	3
1.3. Thesis Outline	3
2. Description Logic Reasoning in Semantic Web	7
2.1. RDF and RDFS	7
2.2. OWL and OWL 2	11
2.3. SPARQL	12
2.4. Description Logic <i>SROIQ</i>	14
2.5. Consistency Checking in Description Logics	19
2.5.1. Tableau algorithm for <i>ALC</i>	20
2.6. Basic Reasoning Services for Description Logics	24
2.7. Expressive Description Logic Queries	28
2.7.1. SPARQL-DL Language	29
2.7.2. Conjunctive ABox Queries	34
2.7.3. Evaluating Conjunctive ABox Queries	35
2.7.4. Distinguished Conjunctive Queries with Negation	40
2.8. Integrity Constraints in OWL	41
3. Ontologies in Information Systems	45
3.1. Ontology-Based Information System Architecture	45
3.1.1. Systems with Generic Architecture	46
3.1.2. Systems with Domain-Specific Architecture	48
3.2. Accessing OWL Ontologies Programmatically	48
3.2.1. Type 1 APIs	49
3.2.2. Type 2 APIs	51
3.3. Relationship of this Thesis to Related Work	53
3.3.1. Ensuring Proper Application-Ontology Contract	53
3.3.2. Providing Expressive Query Language with Efficient Implementation	54
4. Proposed Methodology and Framework	55
4.1. Ontology Persistence Layer	55
4.1.1. Ontology-Object Model Contract	56
4.1.2. Ontology Access Layer	58

4.2.	SPARQL-DL ^{NOT} Language	60
4.2.1.	Optimizing Conjunctive ABox Queries with Undistinguished Variables	62
4.2.2.	Evaluating SPARQL-DL ^{NOT}	65
5.	Software Implementation	77
5.1.	Java OWL Persistence API	77
5.1.1.	Object-Ontology Mapping in Java	80
5.1.2.	Transactional Processing in Java	82
5.2.	SPARQL-DL engine in Pellet and OWL2Query	83
5.2.1.	Evaluation of the Query Engine	83
5.2.2.	Performance of Different Engine Implementations	84
5.2.3.	Performance of the Dynamic Reordering Method	86
5.2.4.	Performance of the Undistinguished Variables Optimizations	87
6.	Use Cases	91
6.1.	StruFail System	91
6.1.1.	System Design	92
6.1.2.	System Usage	96
7.	Conclusions	101
	Bibliography	102
	Acronyms	115
A.	SPARQL-DL Atom Abbreviations	117
B.	SPARQL-DL^{NOT} Atom Cost Estimates	119
C.	Comparing Ontologies using OWLDiff	121
C.1.	OWLDiff Comparison Options	121
C.1.1.	Syntactic Diff	122
C.1.2.	Redundancies	122

List of Figures

1.1.	A roadmap for the work presented in this thesis. Green parts correspond to my novel contributions presented in this thesis, while red/orange parts correspond to related work by other authors. Blue parts are my novel contributions that are linked to the topic of this thesis, but not presented here in detail for the sake of compactness. Yellow parts represent systems that are sketched in this thesis and that were designed and partially implemented by me, while other parts of these systems were implemented under my supervision.	4
2.1.	An RDF graph example. Vertices denote subjects and objects of triples, while edges denote predicates. Vertex and edge URIs are abbreviated using prefixes, in the same way as in the N3 syntax.	9
2.2.	Example RDF document in N3 syntax.	9
2.3.	Example RDF document in RDF/XML syntax.	10
2.4.	Simple SPARQL query to the ontology presented in Example 1. The query asks for all individuals (failures) that affect something (some object). . .	13
2.5.	Initial state of the tableau algorithm.	23
2.6.	Graph G_4 evolved deterministically from G_0 using \sqcap -rule, \exists -rule, \forall -rule. .	24
2.7.	Graphs G_5 and G_6 produced by application of the \sqcup -rule on vertex 0. . .	25
2.8.	Graphs G_7 and G_8 produced by application of the \sqcup -rule on vertex 1. . .	26
2.9.	The difference between undistinguished and distinguished variables is demonstrated on a simple SPARQL-DL query $[PV(?v_1, \text{affects}, !u_2)]$, SPARQL syntax of which is in Figure 2.4. While distinguished variables are bound to individuals (that are always interpreted as domain elements), undistinguished variables (in this case $!u_2$) might be satisfied with a domain element (here $\mathbf{a}_2^{\mathcal{I}}$) that is not an interpretation of any individual in \mathcal{K}	33
2.10.	Query graph G_{Q_1} for Q_1	35
3.1.	Ontology-based system architecture as presented in [1]. (The Figure is reproduced with the permission of its authors.)	47
3.2.	Type 1 APIs comparing to type 2 APIs. Top: Java code generated by Sapphire (type 2). Bottom: The same logic implemented in Jena (type 1) API. Jena code is much less readable (and thus maintainable) than the Sapphire code. Both examples are taken from [2].	50

4.1.	Activity diagram of a transaction run over the Ontology Persistence layer. The business logic access to the front-end layer is represented by the shaded symbols for signal receipt/signal sending.	59
4.2.	Cores extracted from Q_X : $[PV(?v_3, R1, !u_1), PV(!u_1, R2, ?v_2), PV(?v_2, R3, !u_4), PV(!u_1, R4, !u_3), PV(!u_2, R5, !u_3), PV(!u_3, R6, ?v_1), PV(?v_1, R7, c), PV(c, R8, !u_5), PV(?v_2, R9, ?v_1)]$. Dotted arrows represent the edges of G_{Q^v} that build up the cores $\gamma_1, \gamma_2, \gamma_3$, while simple arrows represent edges of G_{Q^D} . The gray rounded rectangles demarcate cores extracted from the Q_X (maximal connected components of G_{Q^v}).	63
4.3.	Exploiting concept hierarchy for down-monotonic variable optimization in LUBM dataset.	75
5.1.	JOPA overall architecture.	78
5.2.	Integrity Constraint Serialization in OWL using RDF/XML syntax. . . .	79
5.3.	An example of generated Java model entity. V , CV are generated classes that serve as vocabularies.	81
6.1.	Fundamental Parts of the StruFail Ontology.	92
6.2.	Part of the Structure Taxonomy.	93
6.3.	Integrity constraint sets for StruFail application. Each vertex corresponds to an OWL 2 DL class and an edge corresponds to an OWL 2 DL object property.	94
6.4.	Exploration of the knowledge base by predefined queries.	97
6.5.	Failure report.	98
C.1.	Protégé Syntactic comparison example. Axioms in one ontology, which do not appear in the other ontology, have green font color in its tree.	123
C.2.	Redundancies comparison example. Axioms that can be inferred from the other ontology by an OWL reasoner, and thus might be redundant, are marked red. Upon selecting such an axiom, a justification of the inference appears in a box at the bottom of the tree.	124

List of Tables

2.1.	<i>SRIOQ</i> axiom examples. F means Failure, S means Structure, hF means hasFactor, af means affects, PS means PillarScour, CB means CharlesBridge, KM means KarluvMost, iFO means isFailureOf.	16
2.2.	Concept and role constructors for <i>SRIOQ</i>	18
2.3.	<i>SRIOQ</i> axioms.	19
2.4.	Completion rules used for expanding a set of \mathcal{ALC} completion graphs. $G = \langle V_G, E_G, L_G \rangle$ is the completion graph chosen in the current iteration.	21
2.5.	<i>SRIOQ</i> reasoner API. The services take as an input an ontology \mathcal{K} , a concept $C_{(i)}$, a role $R_{(i)}$ and an individual \mathbf{a} . The last column denotes the maximal cost of the operation. Due to the complexity of tableau reasoning, the cost is measured in the amount of estimated time spent on tableau algorithm runs. An average time taken by a single tableau algorithm run is denoted as t_{CC}	27
2.6.	Interpretation of SPARQL-DL semi-ground atoms.	32
2.7.	Semantics of selected integrity constraints. Each construct of the form $\bigwedge_{1 \leq i \leq N} X_i$ denotes a list X_1, \dots, X_N . Definition of integrity constraints semantics for other axiom types and other class and property constructors can be found in [3].	42
4.1.	Interpretation of extra SPARQL-DL ^{NOT} semi-ground atoms, complementing the Table 2.6.	61
5.1.	Performance of query evaluation over LUBM(1). <i>O2Q</i> resp. <i>Der</i> mean <i>OWL2Query</i> , resp. <i>Derivo</i> query engines, and <i>P</i> , resp. <i>J</i> means Pellet, resp. JFact tableau reasoner and P_{DM} means Pellet with down-monotonic variable optimization. Next, <i>results</i> denotes number of results of the query. All times are in milliseconds.	86
5.2.	Performance of the dynamic reordering using the Pellet SPARQL-DL engine over the UOB dataset. Each UQ_x denotes a corresponding query from the UOB DL benchmark. <i>atoms*</i> denotes number of query atoms after performing the domain/range simplification, <i>results</i> denotes number of results of the query and <i>static</i> , <i>dynamic</i> and <i>no</i> denote query execution times for these reordering strategies.	87

5.3.	Performance evaluation of the core strategy of the Pellet SPARQL-DL reasoner over the LUBM(1) and UOB(1) DL dataset. The rows labeled <i>simple</i> denote the simple evaluation strategy as described in Example 11, while the rows labeled with <i>core</i> denote the evaluation using cores. The query evaluation took <i>Time</i> ms (without the initial consistency check), <i>results</i> denotes the number of bindings valid for the query and $NB = (N_{IC} + IN N_{IR})/10^3$, where N_{IC} is the count of <i>IC</i> calls and N_{IR} is the count of <i>IR</i> calls.	88
A.1.	SPARQL-DL atom names. Atoms prefixed with * sign are not considered in this thesis, as explained in Section 2.7.1.	117
B.1.	Rough estimates of SPARQL-DL ^{NOT} atom evaluation costs. Notation: $V(\hat{Q}) = V(Q) \setminus B$ is a set of all unbound (i.e. not in B) distinguished variables in Q . Additionally, terms with circumflex denote either a named individual/concept/role, or a variable from B . E.g. \hat{x} means either x (individual) or $x \in B$ (variable bound with unknown individual). Handling of NOT atoms, as well as detailed description of the notation and rationale behind these estimates are presented in Section 4.2.2.	120

1. Introduction

During last years, ontologies [4] have become popular knowledge representation means for wide spectra of problems that deal with incomplete, complex or evolving knowledge, e.g. using ontologies as a background knowledge for automated planning [5], or as a communication language between agents in multi-agent systems [6]. Still, most applications of ontologies are in the field of knowledge modeling for the semantic web [7], [8] – a collection of computer-understandable web pages meaning of which is precisely captured by ontologies.

Plenty of languages for ontology representation have been introduced so far, see Chapter 4 of [9] for an overview. These languages differ in expressiveness, as well as level of formality, ranging from informal, over semi-formal (e.g. OKBC frames [10], topic maps [11]) to formal ones (e.g. KIF [12], conceptual graphs [13]).

However, since 2004, the field has been dominated by formal languages and technologies comprising the so called “Semantic Web Stack” [14], namely W3C¹ recommendations Resource Description Framework (RDF) [15], RDF Schema (RDFS) [16], Web Ontology Language (OWL) [17] and OWL 2 [18]. Significant profiles OWL DL and OWL 2 DL of the latter are built on top of description logic calculus, thus being decidable and having well-defined declarative semantics, details of which are provided in Section 2.4. When referring to ontologies in this thesis, I assume that they are expressed in conformance to these standards.

1.1. Ontologies in Information Systems

As the amount of knowledge published using semantic web principles expands, a need for information systems that would be able to author/manage/search the knowledge grows. Ontology editors (e.g. Protégé [19]) are systems that allow for creating/editing/exploring arbitrary ontologies without making any assumptions about the particular domain of the ontology. However, for an information system to be usable by a non-expert in ontological engineering, domain-dependent business logic (e.g. complex computations) and domain-dependent user interface (e.g. specific visualization) is needed.

Design and maintenance of these, typically object-oriented, domain-dependent information systems that use semantic web ontologies as their data source is non-trivial and has many specifics comparing to the design and maintenance of typical database-backed²

¹Home page of the consortium is at <http://www.w3.org>, cit. 8/10/2010.

²Although various database technologies have been introduced during past decades, including network, hierarchical [20], or object databases [21], from this point on, I consider only relational databases [22] – the main-stream technology backing most production-quality database applications.

information systems.

First, databases accept Closed World Assumption (CWA), which allows them to easily impose integrity constraints on data. For example, if a database prescribes that the surname of a person must be known, and it is not, the closed world assumption is used to infer that no such name exists and constraint violation is reported. Such integrity constraints are important for ensuring data quality but they make the data structure rigid and of limited flexibility. Even a small change (e.g. attribute addition/removal) of the data model requires significant work of an application designer that has to adjust the application business logic accordingly, including updating the application model (typically object model in an object-oriented language), and recompiling the application.

Ontologies, on the other hand, accept Open World Assumption (OWA), thus being suitable for inferring new knowledge in the domain, rather than for data validation. For example, an ontology that prescribes that each person must have a surname, might be consistent no matter whether the surname of a person is known or not – if the surname is not known, its existence (although not necessarily its particular value) is inferred. However, the differences between (open-world) ontological and (closed-world) database descriptions are often neglected by information system designers who make assumptions about ontology structure, as if it was a database schema. As a result, subsequent ontology evolution [23] during application runtime (discussed also in [24]), may violate the original assumptions (called consistency of the ontology and the dependent application in [25]) of the information system designer (e.g. incomplete ontological data with respect to application data structures) at some point. If these assumptions are not formally expressed, it is impossible to guarantee correct functionality of the business logic if the ontology evolves. Thus, on the one hand, ontology changes might cause improper behavior or failure of the application. On the other hand, the application might damage the ontology by producing ontological data that cause its inconsistency.

Second, comparing to databases, OWL ontologies are expressive enough to represent taxonomies (“each woman is a person”) as well as complex relationships between objects, like relation composition (e.g. “being someone’s uncle means being a brother of his/her parent”), cardinality restrictions (e.g. “each person has exactly two arms”), logical negation (e.g. “every person not being a man must be a woman”), etc. This significantly extends the knowledge that can be represented in a declarative manner (and thus reusable in more applications). In order to exploit such expressive descriptions in applications, complex query languages implemented in efficient query engines are necessary. Such queries have to be able to retrieve not only data about particular domain individuals, but also metadata about relationships valid generally in the domain.

This mismatch was faced during the design of an information system aimed at management of structural failure records (for details see Section 6.1). As the domain knowledge is evolving (e.g. new material characteristics, construction technologies and intervention procedures are introduced on the fly) and complex (e.g. transitive part-of relationships on structures, taxonomies of failure/structure types), its implementation using relational database technology would be difficult and hard-to-maintain. This was the motivation (identified also in [25], [26] and [27]) for representing the domain knowledge by means of semantic web ontologies.

1.2. Thesis Contributions

This thesis proposes a methodology for designing information systems that are backed by evolving semantic web ontologies expressed in the OWL 2 language. The overall roadmap of this thesis is depicted in Figure 1.1. Specific novel contributions of this thesis are:

- proposal of a formal contract between an ontology and an object model of an information system; the contract captures the assumptions of the application designer imposed on the ontology and helps to keep the information system up-to-date with respect to the evolving ontology (published in [28]),
- technique for validation of the contract. Once the contract turns out to be invalid, either the application or the ontology has to be adjusted (published in [28]),
- transactional support that aims at keeping the ontology consistent while being accessed by the information system (published in [28]),
- expressive query language SPARQL-DL^{NOT} that is used for evaluation of ontological queries and metaqueries in the information system,
- evaluation and optimization techniques for SPARQL-DL^{NOT}, (published in [29] and [30]),

Techniques presented in this thesis have been implemented in the JOPA system (published in [28]). One of its parts that can be also used as a separate library is the SPARQL-DL^{NOT} engine OWL2Query. The engine is a generalization and extension of the SPARQL-DL [31] engine (published in [29]) that I have designed and implemented in the Pellet reasoner [32].

In addition to the contributions presented in this thesis, some of my other works are closely related to them. First, my SPARQL-DL engine in Pellet has been used for ontology-driven composition of relational data mining workflows (published in [5]). Second, I proposed an ontology comparison scenario, and designed OWLDiff (published in [33]), a system that implements it. OWLDiff was used during the design and development of the prototypical ontology-based information system StruFail (published in [34]), to compare different versions of civil engineering ontologies. For this reason, I shortly sketch ontology comparison scenario as well as OWLDiff in Appendix C. Optimization of OWL inference justification generation algorithms implemented in OWLDiff are not described in this thesis and are published in [35].

1.3. Thesis Outline

The thesis has the following structure. Chapter 2 presents the necessary theoretical background on semantic web ontologies, description logic reasoning and querying. Next, Chapter 3 discusses existing works that relate to the problem of using ontologies in information systems. At the end of the chapter, relation of my work to the existing

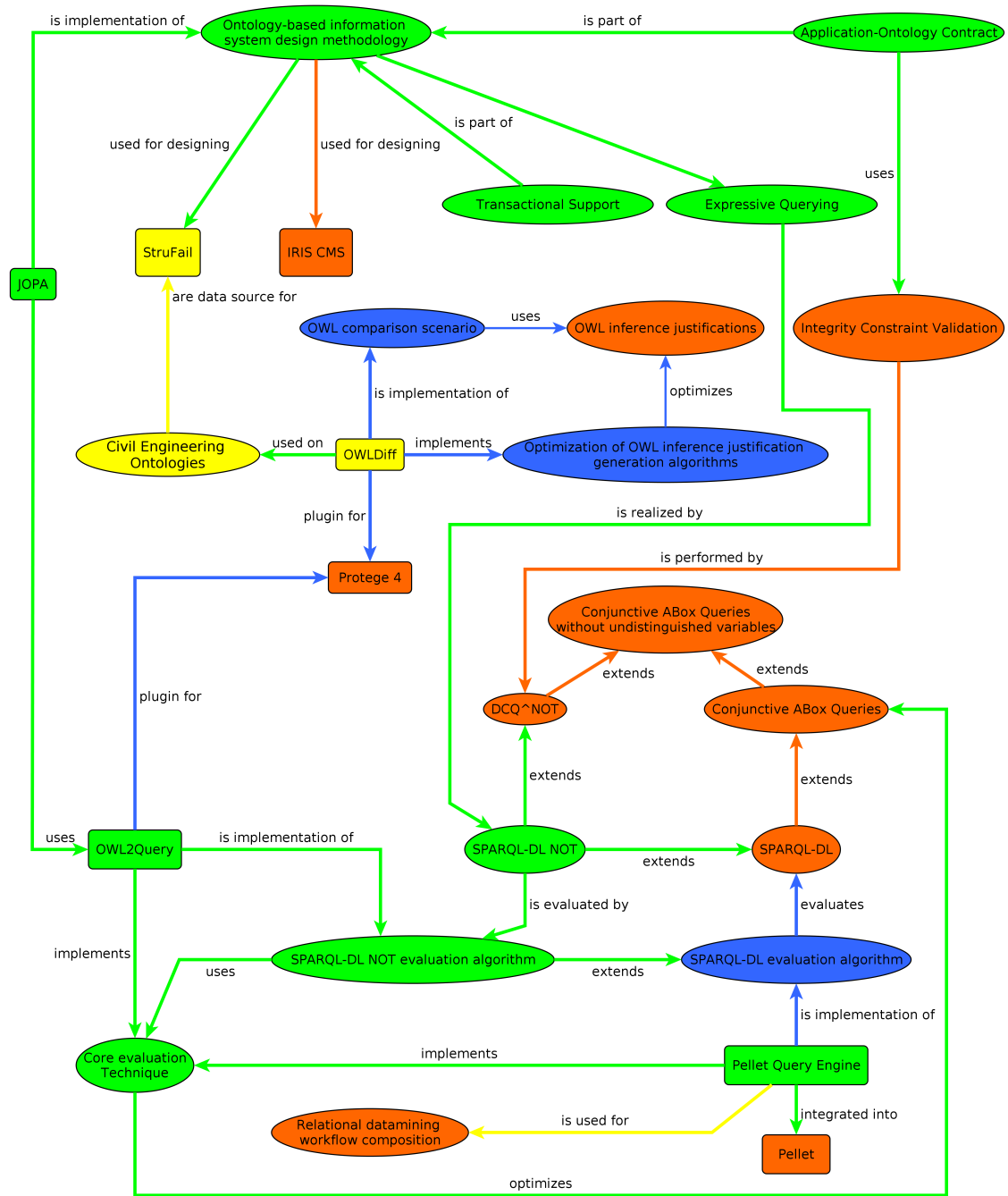


Figure 1.1.: A roadmap for the work presented in this thesis. Green parts correspond to my novel contributions presented in this thesis, while red/orange parts correspond to related work by other authors. Blue parts are my novel contributions that are linked to the topic of this thesis, but not presented here in detail for the sake of compactness. Yellow parts represent systems that are sketched in this thesis and that were designed and partially implemented by me, while other parts of these systems were implemented under my supervision.

approaches is shown. Both Chapter 2 and Chapter 3 build up the knowledge necessary to understand the methodology, techniques and algorithms for using ontologies in information systems proposed in Chapter 4.

Chapter 5 discusses prototypical implementations of the algorithms and together with conducted experiments demonstrate their feasibility. Chapter 6 presents the experience of using the methodology during the design of the StruFail system and its reusability for other approaches.

Partial results of my research related to this thesis were previously published, namely [28] (copyrighted by IEEE), [30] (copyrighted by Springer Verlag), [34], [36] (copyrighted by IGI Global), and [29]. Parts of the thesis are also related to some works authored or coauthored by me, namely [35] (copyrighted by Springer Verlag), [33] (copyrighted by IEEE), and [5] (copyrighted by IEEE).

2. Description Logic Reasoning in Semantic Web

Nowadays, OWL and OWL 2 are standards for representing semantic web ontologies. Their genesis can be tracked from two perspectives. First, OWL stemmed from the principles of semantic web [7] and, as such, can be seen as an expressive semantic annotation language and a successor of RDFS. Second, OWL and OWL 2 are backed by expressive-yet-decidable description logics *SHOIN* and *SROIQ* that have been introduced in last years as a result of intensive research in complexity and decidability of expressive description logics.

I will try to follow both perspectives. First, I shortly introduce RDF, RDFS, OWL and OWL 2, the fundamental technologies of the semantic web stack, as mentioned in Chapter 1, and shortly overview SPARQL Query Language for RDF (SPARQL). The following tutorial-based course of presentation should help the reader understand most important aspects of the declarative knowledge representation by means of semantic technologies, yet it is far from complete – for example I omit rule languages (like Semantic Web Rule Language (SWRL) [37]), that are often used for formalizing complex business logic scenarios, and especially their integration with semantic web ontologies, see e.g. [38], [39], [40], [41].

Next, I will introduce the description logic *SROIQ* that backs OWL 2 and provide details on consistency checking and other basic reasoning services in description logics. These reasoning services are often used for evaluation of expressive queries that are introduced next in this chapter. After an overview of query languages to description logics, I will introduce the SPARQL-DL query language, extension of which is part of the framework presented in Chapter 4, and present evaluation techniques for a subset of SPARQL-DL – conjunctive ABox queries. Last but not least, I recall the integrity constraint proposal for OWL that is used in Chapter 4 as the logical language for expressing contract between an application and an ontology.

2.1. RDF and RDFS

W3C Recommendations RDF and RDFS, standardized in 2004, became one of the first widely accepted technical means for the realization of the semantic web. Due to their mutual dependencies, RDF and RDFS are often considered as one language, denoted as RDF(S). I will not introduce formal definitions here, as they are not important from the perspective of my thesis; interested reader can find formal definitions in the respective W3C Recommendations, i.e. [15], [42] and [16].

RDF describes knowledge about web resources in the form of a *(multi-)graph*. An RDF document consists of *triples* (edges of the graph), each of the form

$$S(\text{subject}) \quad P(\text{predicate}) \quad O(\text{object}).$$

where S and O represent vertices of the graph, while P represents an edge of the graph – a binary relation between S and O. There are three basic objects in RDF:

URIs, representing named resources, e.g.

`http://krizik.felk.cvut.cz/ontologies/2011/example.owl#Failure`, denoted as `f : Failure` in Figure 2.1,

blank nodes, representing anonymous resources, e.g. `_ : q` in Figure 2.1,

literals, representing basic data values (integers, strings, etc.), e.g. `"5"`,

where only Uniform Resource Identifiers (URIs) or blank nodes can serve as subjects of triples and only URIs can serve as predicates of triples. Next, RDF defines a few constructs, like containers (bags, sequences, alternatives) and collections (lists). While containers are open/extensible and can be accessed by index, collections are closeable¹ and can be accessed sequentially. Semantics of RDF is captured by two rule-based entailment regimes – *simple entailment* and *RDF entailment*. These regimes are rather weak and only provide a few rules to interpret triples as a graph (e.g. a predicate of a triple is inferred to be of type `rdf : Property`), or define semantics to blank nodes (e.g. a URI subject of a triple can be generalized to a blank node). In addition to the RDF/XML syntax, that is recommended by authors of RDF for use in applications, RDF can be serialized in other syntaxes, e.g. N-triples that simply list RDF graph edges, or its more compact and better readable extension N3.

Example 1 An RDF graph example is shown in Figure 2.1. This graph corresponds to an RDF document, N3 serialization of which is shown in Figure 2.2 and RDF/XML serialization of which is shown in Figure 2.3.

Note, that a resource can be used in various subject/predicate/object positions in a single RDF document. In this case, `f : isFailureOf` is used in all three positions – as a subject, as a predicate, and as an object.

To understand the meaning of the graph, RDF itself does not help much, as only the predicate `rdf : type` is defined in the RDF vocabulary. The triple

$$f : \text{Watering} \quad \text{rdf : type} \quad f : \text{Failure} .$$

has the meaning “Watering is a Failure”, while the triple

$$_ : q \quad \text{rdf : type} \quad \text{owl : Restriction} .$$

has the meaning that “there is a resource (represented by blank node `_ : q`) of type `owl : Restriction`”.

¹After creation of a *closeable* collection, no more elements can be added to it.

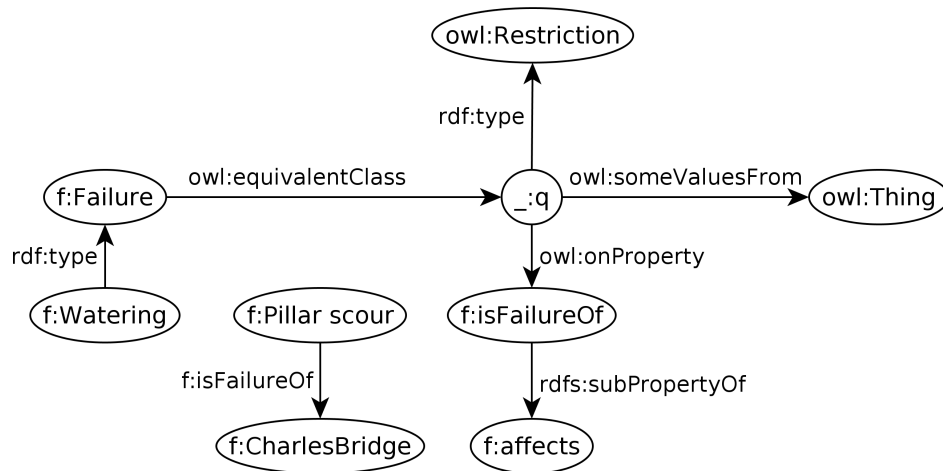


Figure 2.1.: An RDF graph example. Vertices denote subjects and objects of triples, while edges denote predicates. Vertex and edge URIs are abbreviated using prefixes, in the same way as in the N3 syntax.

```

@prefix f: <http://krizik.felk.cvut.cz/ontologies/2011/example.owl#> .
@prefix owl: <http://www.w3.org/2002/07/owl#> .
@prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#> .
@prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#> .

f:Failure owl:equivalentClass
    [ rdf:type owl:Restriction ;
      owl:onProperty f:isFailureOf ;
      owl:someValuesFrom owl:Thing ] .
f:isFailureOf rdfs:subPropertyOf f:affects .
f:Watering rdf:type f:Failure .
f:PillarScour f:isFailureOf f:CharlesBridge .

```

Figure 2.2.: Example RDF document in N3 syntax.

```

<?xml version="1.0"?>
<!DOCTYPE rdf:RDF [
  <!ENTITY rdf "http://www.w3.org/1999/02/22-rdf-syntax-ns#" >
  <!ENTITY rdfs "http://www.w3.org/2000/01/rdf-schema#" >
  <!ENTITY owl "http://www.w3.org/2002/07/owl#" >
]>
<rdf:RDF xmlns="http://krizik.felk.cvut.cz/ontologies/2011/example.owl#"
  xml:base="http://krizik.felk.cvut.cz/ontologies/2011/example.owl"
  xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
  xmlns:rdfs="http://www.w3.org/2000/01/rdf-schema#"
  xmlns:owl="http://www.w3.org/2002/07/owl#">
  <owl:Class rdf:ID="Failure">
    <owl:equivalentClass>
      <owl:Restriction>
        <owl:onProperty rdf:resource="#isFailureOf"/>
        <owl:someValuesFrom rdf:resource="#owl;Thing"/>
      </owl:Restriction>
    </owl:equivalentClass>
  </owl:Class>
  <owl:ObjectProperty rdf:ID="isFailureOf">
    <rdfs:subPropertyOf rdf:resource="#affects"/>
  </owl:ObjectProperty>
  <owl:Thing rdf:ID="PillarScour">
    <isFailureOf rdf:resource="#CharlesBridge"/>
  </owl:Thing>
  <Failure rdf:ID="Watering"/>
</rdf:RDF>

```

Figure 2.3.: Example RDF document in RDF/XML syntax.

RDFS is a schema language for RDF that can serve for building simple ontologies, including class and property definitions and hierarchies (with multiple inheritance), and property domains and ranges. The semantics of RDFS is captured by several inference rules, referred as *RDFS entailment*, see [42].

Example 2 *RDFS gives meaning to the RDF property `rdfs:subPropertyOf` in Figure 2.1. The triple*

`f:isFailureOf rdfs:subPropertyOf f:affects .`

says that “if `f:isFailureOf` relates two resources, then these resources are also related by `f:affects`”. Thus, RDFS entailment allows to infer (among others) that

`f:PillarScour f:affects f:CharlesBridge .`

from the graph in Figure 2.1.

2.2. OWL and OWL 2

OWL, standardized [17] in 2004 and its successor OWL 2, standardized [18] in 2009, aim at representing expressive ontologies on the semantic web. OWL extends RDFS by expressive modeling constructs including boolean operations (intersection, union, complement), existential and universal quantification, number restrictions, and other. Unfortunately, automated reasoning in OWL (and thus also in OWL 2) is undecidable. To cope with decidability and computational complexity, various profiles of OWL and OWL 2 have been defined. OWL has been defined in three variants with decreasing expressiveness:

OWL Full is an extension of RDFS. It offers all features of OWL, which makes it undecidable, as observed in [43].

OWL DL is a decidable sublanguage of OWL Full, corresponding to the description logic $\mathcal{SHOIN}(\mathcal{D})$. For details on $\mathcal{SHOIN}(\mathcal{D})$, see [17].

OWL Lite is a sublanguage of OWL DL with better practical tractability (computational complexity is exponential in time for OWL Lite and even nondeterministic exponential in time for OWL DL, see [44]).

OWL 2 is an extension² of OWL with several expressive modeling constructs, like qualified number restrictions or role chains. Similarly to OWL, also OWL 2 has several variants:

OWL 2 Full is an extension of OWL Full, thus being undecidable.

²OWL 1.1 was used as a previous name for OWL 2.

OWL 2 DL is a decidable extension of OWL DL, corresponding to the description logic $\mathcal{SROIQ}(\mathcal{D})$ that is in turn an extension of $\mathcal{SHOIN}(\mathcal{D})$. Description logic $\mathcal{SROIQ}(\mathcal{D})$ is introduced in Section 2.4.

OWL 2 EL, OWL 2 QL, OWL 2 RL are three tractable profiles of OWL 2 DL. These profiles trade some of OWL 2 DL expressiveness for tractability and thus are suitable for specific application scenarios.

Although syntactic variants of description logic knowledge modeling constructs are the crucial parts of OWL and OWL 2, the specifications define other features for ontology management (ontology importing mechanism, ontology version URIs) or ontology meta-data (annotations on classes, properties, individuals and axioms). Although important from the point of practical usability³, these additional features do not require logical reasoning and thus, whenever no confusion is possible, I will often use the terms OWL 2 DL and $\mathcal{SROIQ}(\mathcal{D})$ interchangeably (and similarly the terms OWL DL and $\mathcal{SHOIN}(\mathcal{D})$).

Example 3 *The RDF document in Example 1 represents a valid OWL 2 DL (and also valid OWL DL) ontology. Thus, an OWL 2 DL agent understands the ontology as a set of description logic axioms (Please refer to the first four axioms in Table 2.1 in Section 2.4 for details on their syntax and semantics):*

Failure $\equiv \exists \text{isFailureOf} \cdot \top$	(α_1)
isFailureOf $\sqsubseteq \text{affects}$	(α_2)
Failure(Watering)	(α_3)
isFailureOf(PillarScour, CharlesBridge)	(α_4)

Axioms α_2 , α_3 and α_4 capture the same meaning as the triples already interpreted by RDF and RDFS, see Section 2.1, while α_1 represents a novel knowledge, not understood by RDFS. Comparing to RDFS entailment, an OWL reasoner is able to infer from the ontology that (i) $f : \text{Watering}$ is related through $f : \text{isFailureOf}$ (and thus also through $f : \text{affects}$) with some (in this case anonymous) object and (ii) classifies $f : \text{PillarScour}$ as a $f : \text{Failure}$. These inferences can be rewritten to RDF triples as

$f : \text{Watering}$	$f : \text{isFailureOf}$	$_ : b$.
$f : \text{Watering}$	$f : \text{affects}$	$_ : b$.
$f : \text{PillarScour}$	rdf : type	$f : \text{Failure}$.

2.3. SPARQL

For RDF ontologies, several query languages appeared during last ten years, but only one won – SPARQL has been standardized [45] in 2008 by W3C and its new version

³E.g. OWL 2 axiom annotations will be used in Chapter 5 for differentiating OWL axioms from OWL integrity constraints

```

PREFIX: <http://krizik.felk.cvut.cz/ontologies/2009/failures.owl#>
SELECT ?x
WHERE {
    ?x :affects _:y .
}

```

Figure 2.4.: Simple SPARQL query to the ontology presented in Example 1. The query asks for all individuals (failures) that affect something (some object).

SPARQL 1.1, extending its predecessor with advanced constructs, like aggregate functions, or path expressions, is currently under standardization process [46].

A basic SPARQL query is formulated using *basic graph patterns*, i.e. generalized RDF graphs containing variables, in addition to URIs, blank nodes and literals. Basic graph patterns can be combined into complex graph patterns using algebraic operations, like UNION (union of two relations), OPTIONAL (analogous to SQL left outer join of two tables/relations), or FILTER (relation row filter). Basic result set of a SPARQL graph pattern is a relation (table), that can be directly outputted (SELECT query type), or transformed to an RDF graph (CONSTRUCT query type) or to boolean true/false result (ASK query type).

I will not introduce full SPARQL syntax and semantics, as they are not directly related to this thesis. Instead, I will document its use on a simple example.

Example 4 *Let's take a SPARQL query in Figure 2.4. The query contains a variable $?x$ that can match URI resources, and a blank node variable $_:y$, that can match any resources existence of which is inferred from the RDF graph. This distinction between variables and blank nodes is similar to the difference of distinguished variables and undistinguished variables defined in Section 2.7.1 (detailed discussion on distinguished and undistinguished variables in SPARQL can be found in [47]).*

Results of this query differ based on the entailment regime that is used for evaluating the SPARQL query. The SPARQL specification allowed to use the RDF, RDFS, and OWL DL entailment regimes, similarly to the different interpretation of RDF graphs shown in Examples 1, 2, and 3. As a result,

- *when evaluating the query using RDF entailment, no binding for $?x$ is returned,*
- *when evaluating the query using RDFS entailment, a single binding for $?x$ is returned: $f : \text{PillarScour}$,*
- *when evaluating the query using OWL DL entailment, two bindings for $?x$ are returned: $f : \text{PillarScour}$, $f : \text{Watering}$.*

2.4. Description Logic \mathcal{SROIQ}

Now, let's switch the view and focus on the details of reasoning in OWL – the description logics. In general, description logics [48] are (typically decidable) subsets of First-Order Predicate Logic (FOPL). Basic building blocks of each description logic are named concepts (corresponding to OWL classes, e.g. `Failure`), named roles (corresponding to OWL object properties, e.g. `isFailureOf`) and individuals (corresponding to OWL individuals, e.g. `CharlesBridge`). Axioms in description logic define generic knowledge about the domain (e.g. concept and property taxonomies), as well as assertions about individuals in the particular world. Each description logic is characterized by a set of available concept, role, and axiom constructs, like boolean operations $\neg C$, $C_1 \sqcap C_2$, $C_1 \sqcup C_2$, universal quantification $\forall R \cdot C$, existential quantification $\exists R \cdot C$ and other.

OWL 2 DL, the most expressive-yet-decidable language from the OWL family, semantically corresponds to the description logic $\mathcal{SROIQ}(\mathcal{D})$, introduced in [49]. Here, I will introduce the \mathcal{SROIQ} description logic that does not support data properties and data types, comparing to $\mathcal{SROIQ}(\mathcal{D})$. Although they are important from the practical point of view, data types do not pose any challenge with respect to the goals of this thesis, but would require extra syntax and notation which would worsen readability significantly. For details on OWL 2 and $\mathcal{SROIQ}(\mathcal{D})$, please refer to [18] and [49].

NOTE: I will often make use of the semantic correspondence between OWL 2 DL and \mathcal{SROIQ} description logic, and reuse \mathcal{SROIQ} syntax for writing OWL axioms. Also, when needed, I will use OWL named individuals instead of \mathcal{SROIQ} individuals, OWL object properties instead of \mathcal{SROIQ} roles and OWL classes instead of \mathcal{SROIQ} concepts. For example, for description of algorithms and techniques next in this chapter, \mathcal{SROIQ} logic is more appropriate, while implementations and applications described in Chapter 5 and Chapter 6 work with OWL ontologies – thus OWL terminology is more appropriate there.

Definition 1 (\mathcal{SROIQ} syntax) A \mathcal{SROIQ} vocabulary is a tuple $V = \langle CN, RN, IN \rangle$, where CN , RN , IN are mutually disjoint, CN is a set of named concepts (denoted $A_{(i)}$), RN is a set of named roles (denoted $R_{(i)}$) and IN is a set of individuals (denoted $a_{(i)}$). A set of all roles is defined as $RN \cup RN^-$, where RN^- is a set of inverse roles defined as $RN^- = \{R^- \mid R \in RN\}$. Expressions of type R^- are not considered – they are always shortened to R .

A \mathcal{SROIQ} theory⁴ $\mathcal{K} = \mathcal{T} \cup \mathcal{R} \cup \mathcal{A}$ is a set of \mathcal{SROIQ} axioms divided into three components: $TBox \mathcal{T}$, $RBox \mathcal{R}$, and $ABox \mathcal{A}$. \mathcal{SROIQ} axioms $\alpha_{\mathcal{T}} \in \mathcal{T}$, $\alpha_{\mathcal{R}} \in \mathcal{R}$ and $\alpha_{\mathcal{A}} \in \mathcal{A}$ have the following form ($C_{(i)} \in S_C$ is a concept, $R_{(i)} \in RN \cup RN^-$ is a role,

⁴Strictly speaking, not each \mathcal{SROIQ} theory represents an ontology. However, since this thesis uses description logics *only* as a formalism for ontology axiomatization, I will often refer to a \mathcal{SROIQ} theory as an *ontology*.

$S_{(i)} \in SN \subseteq RN \cup RN^-$ is a simple role, as defined in [49]):

$$\begin{aligned} \alpha_{\mathcal{T}} \leftarrow & \quad C_1 \sqsubseteq C_2 \quad | \quad C_1 \equiv C_2 \\ \alpha_{\mathcal{R}} \leftarrow & \quad Sym(R) \mid Asy(S) \mid Tra(R) \mid Ref(R) \mid Irr(S) \mid Dis(S_1, S_2) \\ & \quad | \quad R_1 \circ \dots \circ R_n \sqsubseteq R \\ \alpha_{\mathcal{A}} \leftarrow & \quad C(a) \quad | \quad R(a_1, a_2) \end{aligned}$$

where axioms of the type $R_1 \circ \dots \circ R_n \sqsubseteq R$ form a regular hierarchy, as defined in [49]. The set S_C of complex concept descriptions can be constructed in \mathcal{SROIQ} using the following grammar rule:

$$\begin{aligned} C \leftarrow & \top \mid \perp \mid \mathbf{A} \mid \{\mathbf{a}\} \mid \neg C \mid C_1 \sqcap C_2 \mid C_1 \sqcup C_2 \\ & \mid \exists R \cdot C \mid \forall R \cdot C \mid \geq n S \cdot C \mid \leq n S \cdot C \mid \exists S \cdot Self, \end{aligned}$$

where n is a non-negative integer.

□

TBox axioms define concept taxonomies in terms of subsumption (\sqsubseteq) and equivalence (\equiv) of two concepts. Note that either the left-hand side, or the right-hand side, or both sides of $\alpha_{\mathcal{T}}$ might be complex concepts (not only *named concepts*). Thus, in addition to simple hierarchies (e.g. **Church** \sqsubseteq **Structure**), also concept definitions (e.g. α_1 and α_6 in Table 2.1), or even general concept inclusions can be defined (e.g. $\exists \text{isFailureOf} \cdot \text{Church} \equiv (\geq 2 \text{hasFactor} \cdot \text{Factor})$).

RBox axioms define role taxonomies and role characteristics. Axiom $R_1 \circ \dots \circ R_n \sqsubseteq R$ allows to define simple role hierarchies (e.g. **isFailureOf** \sqsubseteq **affects**), as well as complex *role chains* (e.g. **isFailureOf** \circ **partOf** \sqsubseteq **isFailureOf**). Next, common binary relation characteristics can be defined for roles, like (a)symmetry, (ir)reflexivity, disjointness, or transitivity (e.g. $Tra(\text{isPartOf})$).

ABox axioms define the particular relational structure between individuals in terms of their types (e.g. α_3 and α_5 in Table 2.1), or their relationships to other individuals (e.g. α_4 in Table 2.1).

As mentioned in Definition 1, several syntactic restrictions on \mathcal{SROIQ} are placed in [49], to ensure decidability. I will present them only informally here :

- *simple role* is a role that is “not implied by the composition of roles”, i.e. it is either not present on the right hand side of an axiom $R_1 \circ \dots \circ R_n \sqsubseteq R$, or it is, in which case $n = 1$ and R_1 is simple.
- *regular hierarchy* is a set of axioms of the form $R_1 \circ \dots \circ R_n \sqsubseteq R$ that are “not cyclic” and contain only simple roles (or R , but only in either R_1 or R_n positions) on their left-hand side.

	axiom α_i	$\mathcal{I} \models \alpha_i$ iff
α_1	$F \equiv \exists \text{iFO} \cdot \top$	$F^{\mathcal{I}} \subseteq \{x \mid \exists y : \langle x, y \rangle \in \text{iFO}^{\mathcal{I}}\}$ <i>“Each failure is related to an object which it is a failure of (although the object need not be known explicitly).”</i>
α_2	$\text{iFO} \sqsubseteq \text{af}$	$\text{iFO}^{\mathcal{I}} \subseteq \text{af}^{\mathcal{I}}$ <i>“Whenever something is a failure of an object, it also affects it.”</i>
α_3	$F(W)$	$W^{\mathcal{I}} \in F^{\mathcal{I}}$ <i>“Watering is a failure.”</i>
α_4	$\text{iFO}(\text{PS}, \text{CB})$	$\langle \text{PS}^{\mathcal{I}}, \text{CB}^{\mathcal{I}} \rangle \in \text{iFO}^{\mathcal{I}}$ <i>“Pillar scour is a failure of Charles Bridge.”</i>
α_5	$(F \sqcap (\geq 2 \text{hF} \cdot \top))(\text{PS})$	$\text{PS}^{\mathcal{I}} \in F^{\mathcal{I}} \cap \left\{ x \mid \left \{y \mid \langle x, y \rangle \in \text{hF}^{\mathcal{I}}\} \right \geq 2 \right\}$ <i>“Pillar scour is a failure caused by at least two factors.”</i>
α_6	$F \sqsubseteq \forall \text{af} \cdot S$	$F^{\mathcal{I}} \subseteq \{x \mid \forall y : \langle x, y \rangle \in \text{af}^{\mathcal{I}} \implies S^{\mathcal{I}}\}$ <i>“Failures affect only objects.”</i>
α_7	$\{\text{CB}\} \equiv \{\text{KM}\}$	$\text{CB}^{\mathcal{I}} = \text{KM}^{\mathcal{I}}$ <i>“CharlesBridge and KarluvMost are the same objects.”</i>

Table 2.1.: *SRIOQ* axiom examples. F means Failure, S means Structure, hF means hasFactor, af means affects, PS means PillarScour, CB means CharlesBridge, KM means KarluvMost, iFO means isFailureOf.

Semantics of \mathcal{SROIQ} is provided as follows:

Definition 2 (\mathcal{SROIQ} semantics) *The semantics is defined using interpretation $\mathcal{I} = \langle \Delta^{\mathcal{I}}, \cdot^{\mathcal{I}} \rangle$, where $\Delta^{\mathcal{I}}$ is the interpretation domain and interpretation function $\cdot^{\mathcal{I}}$ maps*

- *elements of CN (named concepts) to subsets of $\Delta^{\mathcal{I}}$,*
- *elements of RN (named roles) to subsets of $\Delta^{\mathcal{I}} \times \Delta^{\mathcal{I}}$,*
- *elements of IN (named individuals) to elements of $\Delta^{\mathcal{I}}$.*

The interpretation function is extended to complex concept/role constructors as shown in Table 2.2. For a \mathcal{SROIQ} theory \mathcal{K} ,

- *an interpretation \mathcal{I} **is a model of** \mathcal{K} , denoted as $\mathcal{I} \models \mathcal{K}$, whenever $\mathcal{I} \models \alpha$ for each axiom $\alpha \in \mathcal{K}$. The relation $\mathcal{I} \models \alpha$ is defined in Table 2.3,*
- *\mathcal{K} **is consistent**, if $\mathcal{I} \models \mathcal{K}$ for some interpretation \mathcal{I} ,*
- *an axiom α **is a logical consequence of** \mathcal{K} , denoted as $\mathcal{K} \models \alpha$, if $\mathcal{I} \models \alpha$, whenever $\mathcal{I} \models \mathcal{K}$.*

□

In Tables 2.2 and 2.3, the last column shows the common description logic symbol it is used for the construct, e.g. \mathcal{O} stands for nominal support, or \mathcal{I} stands for⁵ inverse role support in the particular logic. Rows having \mathcal{ALC} in the last column denote constructs that belong to the *attributive language with complements* (\mathcal{ALC}), a subset of \mathcal{SROIQ} introduced in [50] that is considered one of the fundamentals for description logic research in the past decades. This language will be used to demonstrate description logic reasoning in Section 2.5.1.

Syntax of \mathcal{SROIQ} , introduced in this section, is redundant. All ABox axioms can be expressed by TBox axioms with the same meaning, e.g. α_4 can be replaced by $\{\text{PillarScour}\} \sqsubseteq \exists \text{isFailureOf} \cdot \{\text{CharlesBridge}\}$. Another common syntactic sugar is to denote $(\geq n C \cdot R) \sqcap (\leq n C \cdot R)$ as $(= n C \cdot R)$, or to denote $(\geq n R \cdot \top)$ as $(\geq n R)$ and similarly for $=$ and \leq . Also, [49] introduces some other constructs that are expressible (as already shown in [49]) using the syntax introduced here, including negative property assertions, or stating that two individuals are different.

Let's discuss another important aspect of \mathcal{SROIQ} . Analogously to other description logics, \mathcal{SROIQ} is *monotonic*, i.e. all logical consequences of an ontology must hold also after introducing additional axioms to the ontology. Monotonicity of \mathcal{SROIQ} is closely related to OWA, i.e. whatever is not known is not assumed to be false, but might be true or false based on some future knowledge.

⁵Note that in this thesis, the symbol \mathcal{I} is used primarily for denoting an interpretation. Usage of the symbol \mathcal{I} to denote inverse role construct will be stated explicitly each time it is used.

CONCEPT C	INTERPRETATION $C^{\mathcal{I}}$	DESCRIPTION	SYMBOL
\top	$\Delta^{\mathcal{I}}$	(universal concept)	\mathcal{ALL}
\perp	\emptyset	(bottom concept)	\mathcal{ALL}
$\neg C$	$\Delta^{\mathcal{I}} \setminus C^{\mathcal{I}}$	(negation)	\mathcal{ALL}
$C_1 \sqcap C_2$	$C_1^{\mathcal{I}} \cap C_2^{\mathcal{I}}$	(intersection)	\mathcal{ALL}
$C_1 \sqcup C_2$	$C_1^{\mathcal{I}} \cup C_2^{\mathcal{I}}$	(union)	\mathcal{ALL}
$\exists R \cdot C$	$\{a \mid \exists b : \langle a, b \rangle \in R^{\mathcal{I}} \wedge b \in C^{\mathcal{I}}\}$	(full existential quantification)	\mathcal{ALL}
$\forall R \cdot C$	$\{a \mid \forall b : \langle a, b \rangle \in R^{\mathcal{I}} \implies b \in C^{\mathcal{I}}\}$	(value restriction)	\mathcal{ALL}
$\{a\}$	$\{a^{\mathcal{I}}\}$	(nominals)	\mathcal{O}
$\exists S \cdot Self$	$\{a \mid \langle a, a \rangle \in S^{\mathcal{I}}\}$	(self restriction)	\mathcal{R}
$\leq n S \cdot C$	$\left\{ a \mid \left \{b \mid \langle a, b \rangle \in S^{\mathcal{I}} \wedge b \in C^{\mathcal{I}}\} \right \leq n \right\}$	(qualified number restriction)	\mathcal{Q}
$\geq n S \cdot C$	$\left\{ a \mid \left \{b \mid \langle a, b \rangle \in S^{\mathcal{I}} \wedge b \in C^{\mathcal{I}}\} \right \geq n \right\}$	(qualified number restriction)	\mathcal{Q}
ROLE R	INTERPRETATION $R^{\mathcal{I}}$	DESCRIPTION	
$\top^{\mathcal{R}}$	$\Delta^{\mathcal{I}} \times \Delta^{\mathcal{I}}$	(universal role)	
R^{-}	$\{\langle a, b \rangle \mid \langle b, a \rangle \in R^{\mathcal{I}}\}$	(inverse roles)	\mathcal{I}

Table 2.2.: Concept and role constructors for \mathcal{SROIQ} .

Example 5 *To demonstrate open world assumption, consider four consistent \mathcal{SROIQ} ontologies:*

$$\begin{aligned}
\mathcal{K}_1 &= \{\text{Failure}(\text{PillarScour}), \text{Failure} \sqsubseteq \forall \text{isFailureOf} \cdot \text{Structure}\} \\
\mathcal{K}_2 &= \mathcal{K}_1 \cup \{\text{Failure} \sqsubseteq (= 1 \text{ isFailureOf})\}, \\
\mathcal{K}_3 &= \mathcal{K}_2 \cup \{\text{isFailureOf}(\text{PillarScour}, \text{CharlesBridge})\}, \\
\mathcal{K}_4 &= \mathcal{K}_3 \cup \{\text{isFailureOf}(\text{PillarScour}, \text{KarluvMost})\}.
\end{aligned}$$

A logical consequence of \mathcal{K}_3 (and also \mathcal{K}_4) is the axiom $\text{Structure}(\text{CharlesBridge})$ saying that CharlesBridge is an instance of Structure , although \mathcal{K}_3 does not contain this axiom. Similarly, a logical consequence of \mathcal{K}_4 is the axiom $\{\text{CharlesBridge}\} \equiv \{\text{KarluvMost}\}$ saying that CharlesBridge and KarluvMost are the same individuals, although \mathcal{K}_4 does not contain this axiom. On the other hand, although \mathcal{K}_2 does not contain the axiom $\text{isFailureOf}(\text{PillarScour}, \text{CharlesBridge})$, its opposite, i.e. the axiom $\{\text{PillarScour}\} \sqsubseteq \forall \text{isFailureOf} \cdot \neg\{\text{CharlesBridge}\}$ is not a logical consequence of \mathcal{K}_2 . Adding this axiom into \mathcal{K}_3 would cause its inconsistency.

These examples show the benefits of ontologies for “completing” incomplete knowledge (e.g. on the semantic web). However, they are insufficient for applications that need

axiom $\alpha_{\mathcal{T}}$	$\mathcal{I} \models \alpha_{\mathcal{T}}$ iff	DESCRIPTION	SYMBOL
$C_1 \sqsubseteq C_2$	$C_1^{\mathcal{I}} \subseteq C_2^{\mathcal{I}}$	(subsumption)	\mathcal{ALC}
$C_1 \equiv C_2$	$C_1^{\mathcal{I}} = C_2^{\mathcal{I}}$	(equivalence)	\mathcal{ALC}
axiom $\alpha_{\mathcal{R}}$	$\mathcal{I} \models \alpha_{\mathcal{R}}$ iff	DESCRIPTION	
$R_1 \circ \dots \circ R_n \sqsubseteq R$	$R_1^{\mathcal{I}} \circ \dots \circ R_n^{\mathcal{I}} \subseteq R^{\mathcal{I}}$	(role hierarchy)	\mathcal{R}
$Tra(R)$	$\langle a, b \rangle \in R^{\mathcal{I}} \wedge \langle b, c \rangle \in R^{\mathcal{I}} \implies \langle a, c \rangle \in R^{\mathcal{I}}$	(transitivity)	$trans$
$Sym(R)$	$\langle a, b \rangle \in R^{\mathcal{I}} \implies \langle b, a \rangle \in R^{\mathcal{I}}$	(symmetry)	\mathcal{R}
$Asy(S)$	$\langle a, b \rangle \in S^{\mathcal{I}} \implies \langle b, a \rangle \notin S^{\mathcal{I}}$	(asymmetry)	\mathcal{R}
$Ref(R)$	$\forall a : \langle a, a \rangle \in R^{\mathcal{I}}$	(reflexivity)	\mathcal{R}
$Irr(S)$	$\forall a : \langle a, a \rangle \notin S^{\mathcal{I}}$	(irreflexivity)	\mathcal{R}
$Dis(S_1, S_2)$	$S_1^{\mathcal{I}} \cap S_2^{\mathcal{I}} = \emptyset$	(disjointness)	\mathcal{R}
axiom $\alpha_{\mathcal{A}}$	$\mathcal{I} \models_{\mathcal{A}} \text{iff}$	DESCRIPTION	
$C(\mathbf{a})$	$\mathbf{a}^{\mathcal{I}} \in C^{\mathcal{I}}$	(concept assertion)	\mathcal{ALC}
$R(\mathbf{a}_1, \mathbf{a}_2)$	$(\mathbf{a}_1^{\mathcal{I}}, \mathbf{a}_2^{\mathcal{I}}) \in R^{\mathcal{I}}$	(role assertion)	\mathcal{ALC}

Table 2.3.: \mathcal{SROIQ} axioms.

to define the form of data they manipulate – this problem is addressed by integrity constraints introduced in Section 2.8.

2.5. Consistency Checking in Description Logics

There are several approaches that can be used for deciding consistency of an ontology in description logics. First, there are attempts to exploit the analogy of description logics with other logic formalisms. In [51] and [52], an ontology is translated into FOPL and then it is processed by an existing FOPL reasoner. Other approaches try to make use of translation into modal logics or propositional dynamic logics (for both see [48]). Adaptations of well-known resolution techniques for description logics are studied in [53] and [54].

However, current state of the art description logic reasoners (e.g. Pellet [55]) are typically based on a form of a *tableau algorithm*. The main idea of a tableau algorithm can be expressed as “Try to construct a model of the ontology. If a model can be constructed, then the ontology is consistent, otherwise it is not.” In case of description logics, tableau algorithms won over the other approaches due to their transparentness and possibility of efficient optimizations (see Chapter 9 in [48]).

In this section I will introduce the tableau algorithm for a subset of \mathcal{SROIQ} description logic – the attributive language with complements \mathcal{ALC} , introduced in [50]. \mathcal{ALC} allows to use only boolean constructs (\neg, \sqcap, \sqcup) and quantifiers (\forall, \exists) for constructing complex

concepts – it disallows number restrictions, nominals, inverse properties as well as RBox axioms. Detailed description on constructs available in \mathcal{ALC} is shown in Table 2.2 and 2.3 in Section 2.4. Comparing to the tableau algorithm for \mathcal{SROIQ} that is introduced in [49], tableau algorithm for \mathcal{ALC} is more compact and readable, while allowing to use the same optimizations discussed in Section 2.6.

Tableau algorithms aim at constructing a model of an ontology. During the algorithm run, each (possibly infinite) candidate model is represented by a (necessarily finite) *completion graph*. A completion graph is a labeled oriented graph $G = \langle V_G, E_G, L_G \rangle$, where each $x \in V_G$ is labeled with a set $L_G(x)$ of concepts, and each edge $\langle x, y \rangle \in E_G$ is labeled with a set $L_G(\langle x, y \rangle)$ of roles. Furthermore, a completion graph G

- **contains a direct clash**, if $\{A, \neg A\} \subseteq L_G(x)$ for some named concept A , or $\perp \in L_G(x)$, or $\neg \top \in L_G(x)$,
- **is complete w.r.t. to the set J of completion rules**, if no completion rule from J can be applied on it.

2.5.1. Tableau algorithm for \mathcal{ALC}

First, let's consider a tableau algorithm for consistency checking of an \mathcal{ALC} ontology $\mathcal{K} = \mathcal{T} \cup \mathcal{A}$.

1. (PREPROCESSING) All concepts in \mathcal{K} have to be transformed into Negation Normal Form (NNF). This means to “move negation in front of named concepts” using equivalences, like $\neg(C_1 \sqcap C_2) \equiv \neg C_1 \sqcup \neg C_2$, or $\neg(\exists R \cdot C) \equiv \forall R \cdot \neg C$. Furthermore, each equivalence axiom $C_1 \equiv C_2 \in \mathcal{T}$ has to be replaced by two subsumption axioms $C_1 \sqsubseteq C_2$ and $C_2 \sqsubseteq C_1$. See [48] and [50] for more details.
2. (INITIALIZATION) Initial state of the algorithm is $\mathcal{S}_0 = \{G_0\}$, where $G_0 = \langle V_{G_0}, E_{G_0}, L_{G_0} \rangle$ is a completion graph, that “corresponds to \mathcal{A} ”, i.e.
 - V_{G_0} contains all named individuals occurring in some axiom of \mathcal{A} ,
 - E_{G_0} contains all pairs $\langle a_1, a_2 \rangle$ occurring in some $R(a_1, a_2) \in \mathcal{A}$,
 - L_{G_0} labels each vertex (individual) a with a set $\{C \mid C(a) \in \mathcal{A}\}$, and each edge $\langle a_1, a_2 \rangle$ with a set $\{R \mid R(a_1, a_2) \in \mathcal{A}\}$.
3. (CONSISTENCY TEST) Denote the current state as \mathcal{S} . Remove from \mathcal{S} any G that *contains a direct clash*. If $\mathcal{S} = \emptyset$ then return INCONSISTENT.
4. (MODEL TEST) Take arbitrary $G \in \mathcal{S}$ that does not contain a direct clash. If G is *complete* with respect to the completion rules in Table 2.4, then return CONSISTENT.
5. (RULE APPLICATION) Find a completion rule that is applicable on G . Denote the new state \mathcal{S}' created from the current state \mathcal{S} and go to step 2.

\sqsubseteq -rule	if: $(C_1 \sqsubseteq C_2) \in \mathcal{T}$ and $(\neg C_1 \sqcup C_2) \notin L_G(\mathbf{a})$ for some \mathbf{a} that is not blocked. then: $\mathcal{S}' = \mathcal{S} \cup \{G'\} \setminus \{G\}$, where $G' = \langle V_G, E_G, L_{G'} \rangle$. $L_{G'} = L_G$ except $L_{G'}(\mathbf{a}) = L_G(\mathbf{a}) \cup \{\neg C_1 \sqcup C_2\}$.
\sqcap -rule	if: $(C_1 \sqcap C_2) \in L_G(\mathbf{a})$, for some \mathbf{a} that is not blocked and $\{C_1, C_2\} \not\subseteq L_G(\mathbf{a})$. then: $\mathcal{S}' = \mathcal{S} \cup \{G'\} \setminus \{G\}$, where $G' = \langle V_G, E_G, L_{G'} \rangle$. $L_{G'} = L_G$ except $L_{G'}(\mathbf{a}) = L_G(\mathbf{a}) \cup \{C_1, C_2\}$.
\sqcup -rule	if: $(C_1 \sqcup C_2) \in L_G(\mathbf{a})$, for some \mathbf{a} that is not blocked and $\{C_1, C_2\} \cap L_G(\mathbf{a}) = \emptyset$. then: $\mathcal{S}' = \mathcal{S} \cup \{G_1, G_2\} \setminus \{G\}$, where $G_k = \langle V_G, E_G, L_{G_k} \rangle$. $L_{G_k} = L_G$ except $L_{G_k}(\mathbf{a}) = L_G(\mathbf{a}) \cup \{C_k\}$ for $k \in \{1, 2\}$.
\exists -rule	if: $\exists R \cdot C \in L_G(\mathbf{a}_1)$, for some \mathbf{a}_1 that is not blocked, and there is no $\mathbf{a}_2 \in V_G$, such that both $R \in L_G(\langle \mathbf{a}_1, \mathbf{a}_2 \rangle)$ and $C \in L_G(\mathbf{a}_2)$. then: $\mathcal{S}' = \mathcal{S} \cup \{G'\} \setminus \{G\}$, where $G' = \langle V_G \cup \{\mathbf{a}_2\}, E_G \cup \{\langle \mathbf{a}_1, \mathbf{a}_2 \rangle\}, L_{G'} \rangle$. $L_{G'} = L_G$ except $L_{G'}(\mathbf{a}_2) = \{C\}$, $L_{G'}(\langle \mathbf{a}_1, \mathbf{a}_2 \rangle) = \{R\}$.
\forall -rule	if: $\forall R \cdot C \in L_G(\mathbf{a}_1)$, for some \mathbf{a}_1 that is not blocked and there is $\mathbf{a}_2 \in V_G$, such that $R \in L_G(\langle \mathbf{a}_1, \mathbf{a}_2 \rangle)$, but not $C \in L_G(\mathbf{a}_2)$. then: $\mathcal{S}' = \mathcal{S} \cup \{G'\} \setminus \{G\}$, where $G' = \langle V_G, E_G, L_{G'} \rangle$. $L_{G'} = L_G$ except $L_{G'}(\mathbf{a}_2) = L_G(\mathbf{a}_2) \cup \{C\}$.

Table 2.4.: Completion rules used for expanding a set of \mathcal{ALC} completion graphs. $G = \langle V_G, E_G, L_G \rangle$ is the completion graph chosen in the current iteration.

The tableau algorithm evolves a set \mathcal{S} of completion graphs (corresponding to partial candidate models) according to the completion rules in Table 2.4. Whenever a rule application causes a clash in a completion graph, the graph is discarded (which prunes candidate models corresponding to the graph). The algorithm terminates when no more rules can be applied on a clash-free completion graph (ontology is consistent), or when no completion graph remains to be explored (ontology is inconsistent).

To ensure termination of the algorithm it is necessary to detect cycles of the generated model that might occur due to the application of the \sqsubseteq -rule. The cycles are detected using *blocking* that ensures that a completion graph, although representing possibly infinite model, is always finite by ensuring that the completion rules do not generate patterns that “repeat regularly”. The notion of regularity is different for each description logic; for \mathcal{ALC} , so called *subset blocking* [48] is used:

A vertex \mathbf{a}_1 in a completion graph G , but not occurring in \mathcal{A} , is *blocked* by a vertex \mathbf{a}_2 , if there is an oriented path in G from \mathbf{a}_2 to \mathbf{a}_1 and $L_G(\mathbf{a}_1) \subseteq L_G(\mathbf{a}_2)$.

The algorithm does not prescribe the order, in which the rules are selected. Of course, this can significantly influence performance. E.g. non-deterministic rules (\sqcup -rule in case of \mathcal{ALC}) *should* be performed only when no other rule is applicable, to prevent generating additional completion graphs, all of which need to be tested in

CONSISTENCY/MODEL TEST steps of the algorithm. As shown in [44], complexity of \mathcal{ALC} consistency checking is exponential in time (ExpTime-complete) and complexity of \mathcal{SROIQ} even nondeterministic exponential in time (NExpTime). To cope with the complexity, many efficient optimizations of tableau algorithms have been proposed for description logics, as presented in Chapter 9 of [48].

Correctness and Completeness I will not repeat the full proof of correctness and completeness of the algorithm, that was already presented in [48] and [50]. I only sketch main rationale behind the algorithm that helps to better understand its idea. Correctness is a direct consequence of the semantics of completion rules. E.g. if there was a model \mathcal{I} corresponding to G and $A_1 \sqcap A_2 \in L_G(a)$ for some a , then, following Section 2.4, $a^{\mathcal{I}} \in (A_1 \sqcap A_2)^{\mathcal{I}}$ and $a^{\mathcal{I}} \in A_1^{\mathcal{I}} \cap A_2^{\mathcal{I}}$. This is ensured by putting both A_1 and A_2 into the $L'_G(a)$ by the \sqcap -rule. For the other direction and other rules the idea is similar.

Completeness is shown by constructing a canonical model \mathcal{I} of \mathcal{K} for each complete completion graph that does not contain a direct clash, as follows:

- The interpretation domain $\Delta^{\mathcal{I}}$ contains all graph vertices,
- for each named concept A we define $A^{\mathcal{I}} = \{a \mid A \in L_G(a)\}$,
- for each named role R we define $R^{\mathcal{I}} = \{\langle a_1, a_2 \rangle \mid R \in L_G(\langle a_1, a_2 \rangle)\}$,

Induction along the axiom types and complex concept structure shows that it is indeed a model of the original ontology. E.g. if $C(a) \in \mathcal{K}$, then $a^{\mathcal{I}} \in C^{\mathcal{I}}$ must hold for any complete graph G : (i) if $C = A$ is a named concept, then indeed $a^{\mathcal{I}} \in A^{\mathcal{I}}$ because $A \in L_{G_0}(a)$ (this is, how G_0 was constructed in the INITIALIZATION step of the algorithm) and $L_{G_0}(a) \subseteq L_G(a)$, (ii) if $C = A_1 \sqcap A_2$ where both A_1 and A_2 are named concepts, then $a^{\mathcal{I}} \in A_1 \sqcap A_2^{\mathcal{I}}$ and thus $a^{\mathcal{I}} \in A_1^{\mathcal{I}}$ and $a^{\mathcal{I}} \in A_2^{\mathcal{I}}$ because $\{A_1, A_2\} \subseteq L_{G'}$ where G' is a completion graph that resulted from application of the \sqcap -rule. If this rule was not applied, then $G \supseteq G'$ is not complete, which contradicts our assumption. For the other axiom types and concept constructs the idea is similar.

In case of \mathcal{ALC} , there is a correspondence between a model and a complete completion graph simple, as presented. However, tableau algorithms for expressive description logics require more complex structures and more complex transformations to achieve such correspondence, see e.g. [48] or [49] for more details.

□

Example 6 *Let's check consistency of an \mathcal{ALC} ontology $\mathcal{K} = \{\alpha\}$, where α is the axiom $C(\text{PillarScour})$ and C is*

$$(\exists \text{isFailureOf} \cdot \text{Column} \sqcap \exists \text{isFailureOf} \cdot \text{Pillar} \sqcap \neg \exists \text{isFailureOf} \cdot (\text{Pillar} \sqcap \text{Column}))$$

*This axiom says that **PillarScour** is failure of some column and it is a failure of some pillar but it is not a failure of an object that is a column and a pillar at the same time.*

The first step is to transform the complex concept into NNF. This produces α_2 that is $C_2(\text{PillarScour})$ where C_2 is

$$(\exists \text{isFailureOf} \cdot \text{Column} \sqcap \exists \text{isFailureOf} \cdot \text{Pillar} \sqcap \forall \text{isFailureOf} \cdot (\neg \text{Pillar} \sqcup \neg \text{Column})),$$

where α_2 is semantically equivalent with α i.e. $\{\alpha\} \models \{\alpha_2\}$ and $\{\alpha_2\} \models \{\alpha\}$.

The initial state of the algorithm is $\mathcal{S}_0 = \{G_0\}$, where graph

$$G_0 = \langle \{\text{PillarScour}\}, \emptyset, \{\text{PillarScour} \mapsto \{C_2\}\} \rangle$$

is shown⁶ in Figure 2.5. At this point, the algorithm passes four sequences of the steps

<p>"PillarScour"</p> <hr/> <p>$((\exists \text{isFailureOf} \cdot \text{Pillar}) \sqcap (\exists \text{isFailureOf} \cdot \text{Column}) \sqcap (\forall \text{isFailureOf} \cdot (\neg \text{Column} \sqcup \neg \text{Pillar})))$</p>
--

Figure 2.5.: Initial state of the tableau algorithm.

CONSISTENCY TEST \rightarrow *MODEL TEST* \rightarrow *RULE APPLICATION* of the tableau algorithm, that can be denoted as a state evolution during the step *RULE APPLICATION* (the label over the arrow denotes the rule that was used):

$$\{G_0\} \xrightarrow{\sqcap\text{-rule (twice)}} \{G_1\} \xrightarrow{\exists\text{-rule}} \{G_2\} \xrightarrow{\exists\text{-rule}} \{G_3\} \xrightarrow{\forall\text{-rule}} \{G_4\},$$

where G_4 is depicted in Figure 2.6.

So far, only deterministic rules (i.e. those that do not increase the number of completion graphs) have been used. Looking carefully at Figure 2.6, it is clear that the only rule that remains applicable is the \sqcup -rule. The rule can be applied on the concept $(\neg \text{Column} \sqcup \neg \text{Pillar})$ in the label of either vertex 0 or 1. Picking e.g. 0 and applying \sqcup -rule produces a new state $\{G_5, G_6\}$ depicted in Figure 2.7. Graph G_5 contains a direct clash, as Column and $\neg \text{Column}$ is in the label of vertex 0, and thus G_5 is discarded, as it cannot be transformed to a model (e.g. the canonical model defined in the Correctness and Completeness paragraph above). Thus, G_6 is picked and \sqcup -rule is applied, which results in a new state $\{G_7, G_8\}$, as shown in Figure 2.8.

While G_7 contains a direct clash in vertex 1, completion graph G_8 is complete with respect to the \mathcal{ALC} completion rules and does not contain a direct clash. Thus, a canonical model $\mathcal{I}_1 = \langle \Delta^{\mathcal{I}_1}, \mathcal{I}_1 \rangle$ can be constructed from G_8 as follows:

$$\begin{aligned} \Delta^{\mathcal{I}_1} &= \{\text{PillarScour}, 0, 1\}, \\ \text{isFailureOf}^{\mathcal{I}_1} &= \{\langle \text{PillarScour}, 0 \rangle, \langle \text{PillarScour}, 1 \rangle\}, \\ \text{Pillar}^{\mathcal{I}_1} &= \{1\}, \\ \text{Column}^{\mathcal{I}_1} &= \{0\}, \\ \text{PillarScour}^{\mathcal{I}_1} &= \{\text{PillarScour}\}. \end{aligned}$$

⁶Visualization of completion graphs in this example has been created by my implementation of tableau reasoner available at <http://krizik.felk.cvut.cz/km/dl>, cit. 12/1/2012.

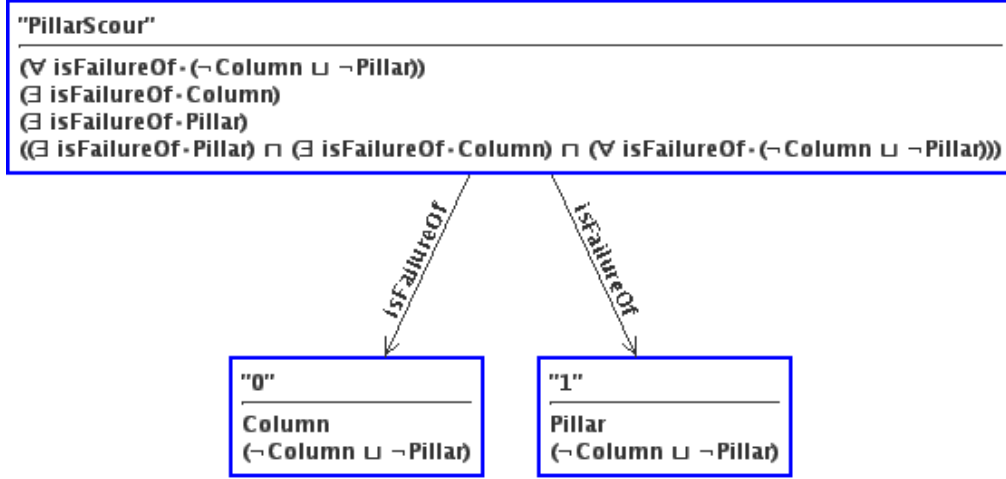


Figure 2.6.: Graph G_4 evolved deterministically from G_0 using \sqcap -rule, \exists -rule, \forall -rule.

\mathcal{I}_1 is not the only interpretation of \mathcal{K} - another model of \mathcal{K} might be \mathcal{I}_2 , for which $\text{Column} = \{0, \text{PillarScour}\}$ and coincides with \mathcal{I}_1 on the rest. This documents the fact that a complete completion graph corresponds to one (canonical) model, but might correspond to other models as well.

2.6. Basic Reasoning Services for Description Logics

Last section demonstrated consistency checking in description logics on a tableau algorithm for \mathcal{ALC} , a subset of OWL DL. Although important, and typically also most computationally demanding, consistency checking is not sufficient in practical applications. Additionally, description logic reasoners support other inference services, ranging from basic queries (discussed in this section) to expressive queries and metaqueries (discussed in Section 2.7.1). Although all of such queries make use of a (typically tableau) consistency checking in their core, wide spectra of optimizations make reasonable to take them as specialized services.

Basic services typically include

basic TBox queries like concept subsumption (or equivalence/disjointness/unsatisfiability checking, etc.), e.g. whether

$$\mathcal{K} \models \exists \text{hasPart} \cdot \text{Pillar} \sqsubseteq \text{Bridge}$$

“Are objects with at least one pillar always bridges?”

basic RBox queries like role subsumption (or equivalence/disjointness checking, etc.), e.g. whether

$$\mathcal{K} \models \text{isFailureOf} \sqsubseteq \text{affects}$$

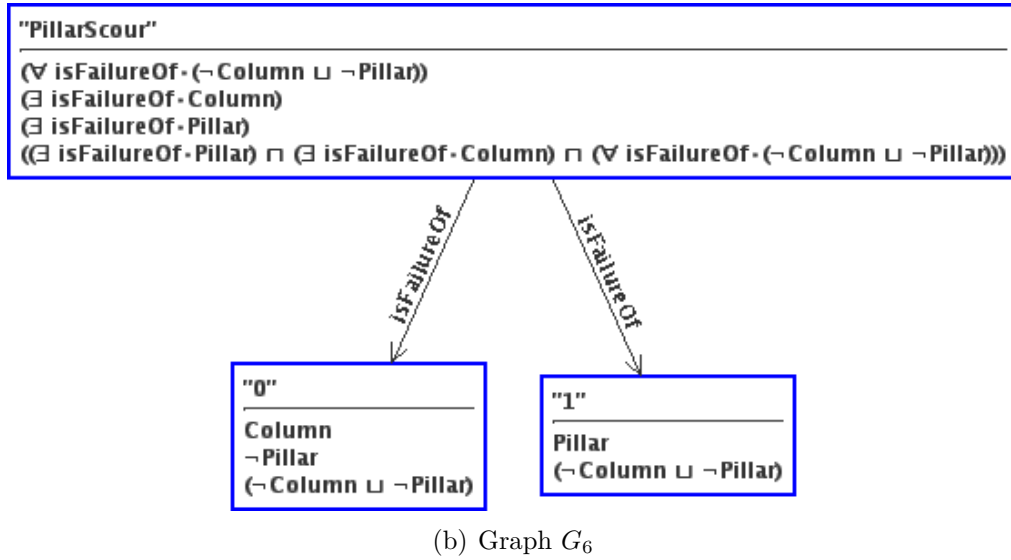
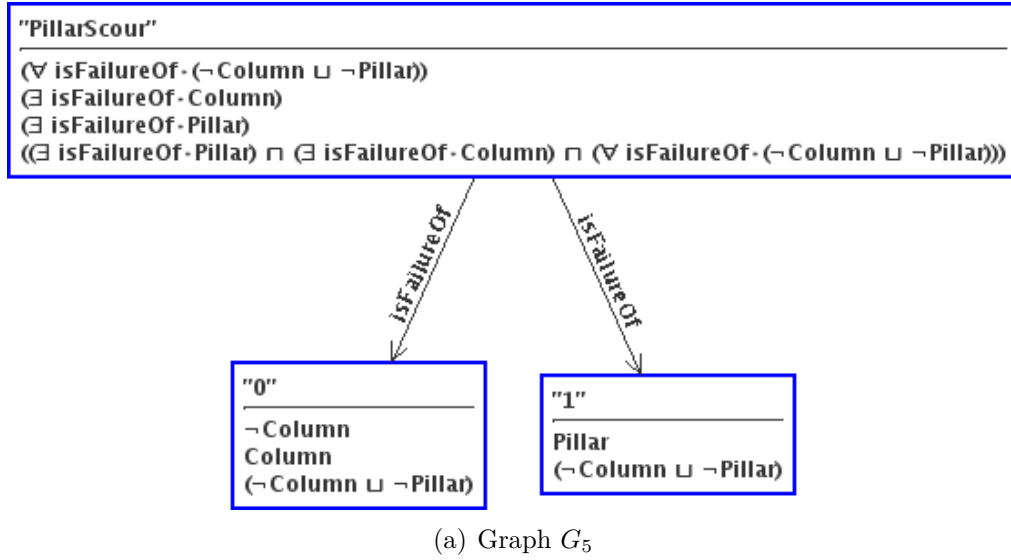


Figure 2.7.: Graphs G_5 and G_6 produced by application of the \sqcup -rule on vertex 0.

“Do failures of objects always affect these objects ?”

basic ABox queries like instance checking (or role checking, instance retrieval, class retrieval, etc.), e.g. whether

$$\mathcal{K} \models (\geq 5 \text{ hasPart} \cdot \text{Pillar})(\text{CharlesBridge})$$

“Does Charles Bridge contain at least five pillars ?”

As shown in [48] and [56], in description logics providing full concept negation construct (both \mathcal{ALC} and \mathcal{SROIQ} satisfy this), basic TBox and ABox inference services can be

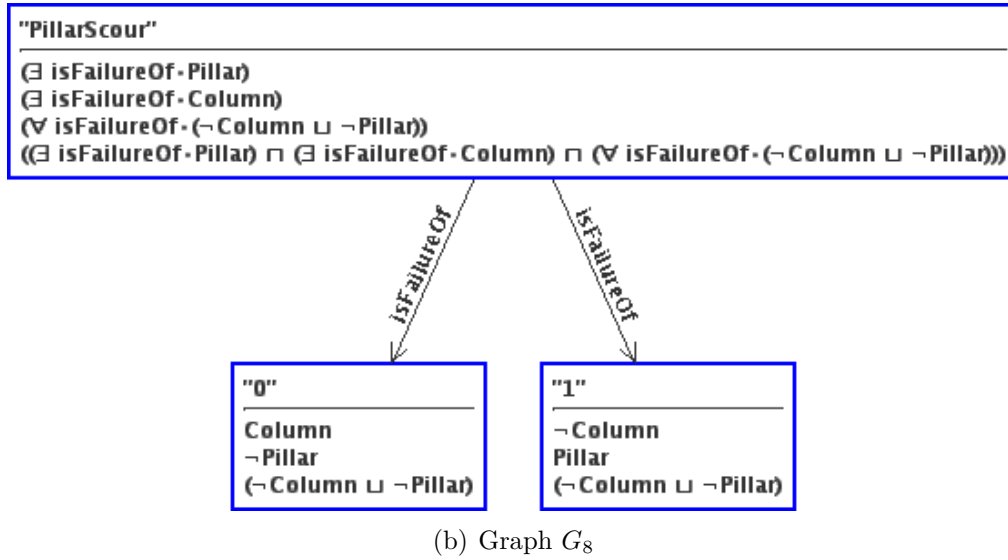
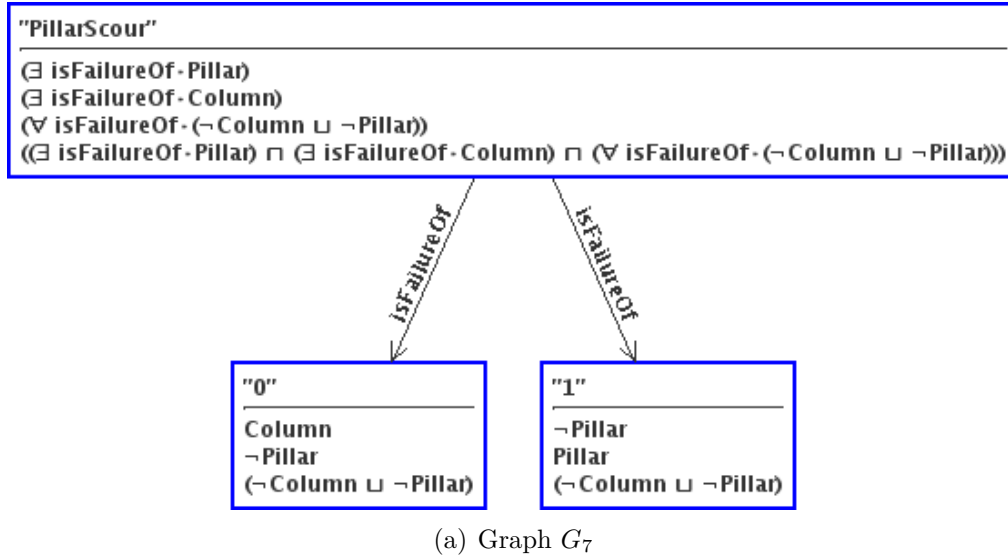


Figure 2.8.: Graphs G_7 and G_8 produced by application of the \sqcup -rule on vertex 1.

reduced to one or more consistency checks. E.g. for \mathcal{K} consistent, $\mathcal{K} \models C(\mathbf{a})$ holds whenever $\mathcal{K} \cup \{(\neg C)(\mathbf{a})\}$ is inconsistent. For more details on basic inference problems and their reduction, see [48].

Based on this, for the purpose of this thesis, I take a *SRIOQ* reasoner as a black box, providing a set of services defined in Table 2.5. The introduced set of operations is not minimal, as all of them could be replaced by one or more CC calls, as discussed above. Still, it is small enough to easily fit to the major Java-based APIs for OWL 2: OWLAPI [57], Jena [58], or reasoner integration API OWLLink [59], as well as native APIs of major OWL 2 reasoners, e.g. Pellet [55], JFact [60], or HermiT [61] (see [62] for an extensive list of OWL 2 reasoner implementations).

	operation o	return value	$\chi(o)$
ABox	$CC(\mathcal{K})$	$true$, iff \mathcal{K} is consistent	$1 \cdot t_{CC}$
	$IC(\mathcal{K}, C, \mathbf{a})$	$true$, iff $\mathcal{K} \models C(\mathbf{a})$	$1 \cdot t_{CC}$
	$IR(\mathcal{K}, C)$	all $\mathbf{a} \in IN$, s.t. $\mathcal{K} \models C(\mathbf{a})$	$ IN \cdot t_{CC}$
	$CR(\mathcal{K}, \mathbf{a})$	all $\mathbf{A} \in CN$, s.t. $\mathcal{K} \models \mathbf{A}(\mathbf{a})$	$ CN \cdot t_{CC}$
TBox	$SUBC(\mathcal{K}, C)$	all $\mathbf{A} \in CN$, s.t. $\mathcal{K} \models \mathbf{A} \sqsubseteq C$	$ CN \cdot t_{CC}$
	$SUPERC(\mathcal{K}, C)$	all $\mathbf{A} \in CN$, s.t. $\mathcal{K} \models C \sqsubseteq \mathbf{A}$	$ CN \cdot t_{CC}$
	$ISUBC(\mathcal{K}, C_1, C_2)$	true iff $\mathcal{K} \models C_1 \sqsubseteq C_2$	$1 \cdot t_{CC}$
	$EQC(\mathcal{K}, C)$	all $\mathbf{A} \in CN$, s.t. $\mathcal{K} \models \mathbf{A} \equiv C$	$ CN \cdot t_{CC}$
RBox	$ISEQC(\mathcal{K}, C_1, C_2)$	true iff $\mathcal{K} \models C_1 \equiv C_2$	$1 \cdot t_{CC}$
	$SUBP(\mathcal{K}, R)$	all $\mathbf{R} \in RN$, s.t. $\mathcal{K} \models \mathbf{R} \sqsubseteq R$	$ RN \cdot t_{CC}$
	$SUPERP(\mathcal{K}, R)$	all $\mathbf{R} \in RN$, s.t. $\mathcal{K} \models R \sqsubseteq \mathbf{R}$	$ RN \cdot t_{CC}$
	$ISUBP(\mathcal{K}, R_1, R_2)$	true iff $\mathcal{K} \models R_1 \sqsubseteq R_2$	$1 \cdot t_{CC}$
	$EQP(\mathcal{K}, R)$	all $\mathbf{R} \in RN$, s.t. $\mathcal{K} \models \mathbf{R} \equiv R$	$ RN \cdot t_{CC}$
	$ISEQP(\mathcal{K}, R_1, R_2)$	true iff $\mathcal{K} \models R_1 \equiv R_2$	$1 \cdot t_{CC}$

Table 2.5.: \mathcal{SROIQ} reasoner API. The services take as an input an ontology \mathcal{K} , a concept $C_{(i)}$, a role $R_{(i)}$ and an individual \mathbf{a} . The last column denotes the maximal cost of the operation. Due to the complexity of tableau reasoning, the cost is measured in the amount of estimated time spent on tableau algorithm runs. An average time taken by a single tableau algorithm run is denoted as t_{CC} .

The last column in Table 2.5 shows maximal number of consistency checks a tableau reasoner has to perform to evaluate the operation. The set of operations has been selected in such a way that the actual number of consistency checks (that are decidable but at-least-exponential, as shown in Section 2.5.1), is typically much smaller due to the optimizations sketched in the rest of this section and described in detail in [48] and [63].

Let's shortly sketch optimizations of basic ABox queries first, as they are more elaborated in literature with respect to conjunctive query answering. As shown in Section 2.5.1, by applying *completion rules*, a tableau algorithm evolves a set of *completion graphs* (see [63] and [64]), that correspond to potential models of the ontology. **Obvious non-instances using completion** optimization of IC, IR and CR refuses each $C(\mathbf{a})$ inference that is in clash with *some* complete completion graph G , i.e. whenever G contains a vertex \mathbf{a} labeled with $\neg C$, and similarly for $R(\mathbf{a}_1, \mathbf{a}_2)$. E.g. during a call to $IC(C, \mathbf{a})$, the reasoner first looks into each completion G . If G contains a vertex \mathbf{a} that has a label $\neg C$, the reasoner rejects the entailment without any additional tableau consistency check.

Another optimization makes use of a *precompletion*, which is a completion graph that was obtained only by application of deterministic rules (in case of \mathcal{ALC} all but \sqcup -rule). Precompletion (e.g. G_0 to G_4 in Example 6) represents a common part of all models of the ontology. In other words, all complete completion graphs (that can be transformed to all possible models of the ontology) contain each precompletion as their subgraph. The most

informative precompletion is the “largest ones”, i.e. precompletion (e.g. G_4 in Example 6) that contains no other precompletion as its subgraph. **Obvious instances using precompletion** optimization of IC, IR and CR makes use of the largest precompletion as a cache for inferences of the form $C(\mathbf{a})$ and $R(\mathbf{a}_1, \mathbf{a}_2)$ that are valid in all models of \mathcal{K} . E.g. during a call to $\text{IC}(C, \mathbf{a})$, the reasoner first looks into the precompletion G . If G contains a vertex \mathbf{a} that has a label C (i.e. $C \in L_G(\mathbf{a})$), the reasoner approves the entailment without any additional tableau consistency check.

In addition to caching completion and precompletion information, IR and CR can be further optimized using methods presented in [63], namely *binary instance retrieval* and its variants. A naive way of finding all instances of a concept C is to perform an instance check $\mathcal{K} \models C(\mathbf{a})$ for each individual \mathbf{a} mentioned in \mathcal{K} (linear instance retrieval). *Binary instance retrieval* optimization tries to reduce the number of instance checks by checking many individuals being instances of C during a single tableau algorithm run using the divide and conquer strategy.

As for the TBox and RBox operations, these make use of concept/role taxonomies that are cached during ontology classification by the reasoner, as described in [48]. In a tableau reasoner, *told concept/role taxonomy* can be constructed from all $C_1 \sqsubseteq C_2$ (resp. $R_1 \sqsubseteq R_2$) axioms that makes use of the transitivity of the \sqsubseteq and \equiv relations. Although this taxonomy is incomplete, many named concepts (resp. roles) can be immediately decided to be part of the result of the $\text{SUBC}(\mathcal{K}, C)$ operation (resp. $\text{SUBP}(\mathcal{K}, R)$), and also other TBox/RBox operations in Table 2.5, without additional tableau consistency checks.

2.7. Expressive Description Logic Queries

Basic reasoning services were just a first step towards queries that can be practically used in Ontology-based Information System (OIS), introduced later in Chapter 3. Thus, expressive query languages with varying features, and their implementations, were introduced during last few years.

The very first common expressive language for ontology queries were *conjunctive ABox queries*, that allow to retrieve individuals from an ontology, thus being similar to select-project-join queries into relational databases. I will discuss conjunctive ABox queries, as well as their extension with negation as failure called Distinguished Conjunctive Queries with Negation (DCQ^{NOT}), later in this Section.

However, comparing to relational databases, OWL ontologies contain also significant amount of metaknowledge for expressing taxonomies and complex characteristics of classes (e.g. disjointness, equivalence) or properties (e.g. transitivity, functionality). To address these constructs, query languages SPARQL-DL, SQWRL ([65]) and OWL-SAIQL ([66]) appeared during last years that allow evaluating mixed ABox, TBox and RBox queries to retrieve individuals, classes, and properties.

SQWRL is a SWRL-based query language [37] that allows to pose queries by combining OWL axioms and SWRL atoms, in which individual/class/property names can be replaced by distinguished variables. Comparing to SPARQL-DL and OWL-SAIQL, SQWRL allows

also to formulate aggregate queries.

OWL-SAIQL was another proposal for expressive queries to OWL. It allows for retrieving named classes as well as complex class descriptions, comparing to SQWRL and SPARQL-DL.

SPARQL-DL allows for both distinguished variables and undistinguished variables in queries and provides native SPARQL [45] syntax. Distinguished variables bind individuals/classes/properties that are returned in the result set, while undistinguished variables match domain elements that need not be interpretations of individuals in the queried ontology. Thus, undistinguished variables are handling OWL semantics (and thus also OWL DL entailment for SPARQL) closely, being able to match inferred domain elements that are not represented by individuals. Specifying a variable as undistinguished, rather than distinguished, might significantly influence the result set of a query, as shown in Example 7 in Section 2.7.1.

Support for both undistinguished and distinguished variables were the principal reasons to base my work in Chapter 4 on SPARQL-DL. Another reason was that SPARQL syntax for SPARQL-DL simplifies introduction of OWL query semantics in already existing semantic web applications backed by RDF and SPARQL, as no transformations between query syntaxes are needed for the query engine users.

In this section I recall the two expressive query languages that generalize basic reasoning services and that are used in this thesis:

1. SPARQL-DL queries, and
2. DCQ^{NOT} queries,

both of which play important role in the methodology and framework introduced in Chapter 4. DCQ^{NOT} queries are used to formalize semantics of OWL integrity constraints, as shown in Section 2.8, while $SPARQL-DL^{NOT}$, an extended version of SPARQL-DL, is used for posing expressive queries to OWL ontologies. Other expressive query languages are discussed in Chapter 3. I will start with the definition of SPARQL-DL and introduce also *conjunctive ABox queries* as their significant subset. DCQ^{NOT} queries are introduced next.

2.7.1. SPARQL-DL Language

SPARQL-DL [31] was introduced as an expressive query and metaquery language to OWL DL ontologies. Originally, it was designed as an advanced OWL-compliant entailment regime for the SPARQL language, thus realizing the OWL DL entailment of basic graph patterns, as discussed in Section 2.3. As a result, SPARQL syntax for basic graph patterns was originally used for SPARQL-DL queries. Several alternative syntaxes have been suggested for SPARQL-DL since that time. In [67], a new syntax called *Terp* that combines the SPARQL syntax with simplified syntax for OWL class constructs, was introduced. This syntax was meant as a more readable alternative for SPARQL users. Another syntax called *SPARQLAS* that makes use of templates similar to OWL

functional syntax is introduced in [68]. Here, I will use the original SPARQL-DL abstract syntax defined below exclusively.

Definition 3 (SPARQL-DL abstract syntax) *Let's have a \mathcal{SROIQ} ontology \mathcal{K} , with vocabulary $V = \langle CN, RN, IN \rangle$, as defined in Section 2.4. Next, consider a set \mathcal{V}_{var} of distinguished variables and a set \mathcal{V}_{bnode} of undistinguished variables, both disjoint from $CN \cup RN \cup IN$. We define a query atom q using the following expansion⁷:*

$$\begin{aligned} q \leftarrow & \text{Ty}(a, c) \mid \text{PV}(a_1, r, a_2) \mid \text{SA}(a_1, a_2) \\ & \mid \text{DF}(a_1, a_2) \mid \text{SCO}(c_1, c_2) \mid \text{EC}(c_1, c_2) \mid \text{DW}(c_1, c_2) \\ & \mid \text{CO}(c_1, c_2) \mid \text{SPO}(r_1, r_2) \mid \text{EP}(r_1, r_2) \\ & \mid \text{IO}(r_1, r_2) \\ & \mid \text{Fun}(r) \mid \text{IFun}(r) \mid \text{Sym}(r) \mid \text{Trans}(r), \end{aligned}$$

where $a_{(i)} \in IN \cup \mathcal{V}_{var} \cup \mathcal{V}_{bnode}$, $c_{(i)} \in S_C \cup \mathcal{V}_{var}$, $r_{(i)} \in RN \cup \mathcal{V}_{var}$,

- A SPARQL-DL query Q is a list $[q_1, \dots, q_n]$ of query atoms q_i , interpreted as their conjunction.
- Set $V(Q) \subseteq \mathcal{V}_{var}$ (resp. $U(Q) \subseteq \mathcal{V}_{bnode}$, resp. $I(Q) \subseteq IN$) consists of all distinguished variables (resp. undistinguished variables, resp. individuals) that appear as arguments of some query atom of Q .
- A semi-ground query is a query Q with $V(Q) = \emptyset$. Each $q \in Q$ is called a semi-ground atom.

□

This definition differs slightly from the definitions in [31]. For the same reasons as in Section 2.4, this definition omits data properties and parts of OWL that are not backed by logical reasoning, that were presented in [31]. Thus **Annotation**, **ObjectProperty** and **DataProperty** query atoms are omitted, and **PropertyValue** query atom can only connect an individual/variable to another individual/variable, and not to data literal. Another difference is that in the original definition $a_{(i)}$ is picked from the set of $IN \cup RN \cup CN \cup \mathcal{V}_{var} \cup \mathcal{V}_{bnode}$, instead of $IN \cup \mathcal{V}_{var} \cup \mathcal{V}_{bnode}$. The original definition reflects OWL 2 mechanism for syntactic name overloading called *punning* that is not available in \mathcal{SROIQ} . This mechanism is purely syntactic means that allows for using the same name for two different entities (e.g. for an individual **Car** and a concept **Car**), but with different interpretation for each of the two meanings (e.g. $\text{Car}^{\mathcal{I}}$ is a domain element whenever the occurrence of **Car** is used in the position of an individual, and it is a set whenever **Car** is used in the position of a concept).

⁷To keep descriptions compact and whenever it will not compromise readability, I will use shortened versions of the original atom names, e.g. **Type** will be abbreviated as **Ty**, atom **PropertyValue** as **PV**, or **InverseFunctional** as **IFun**, etc. Full atom names are listed in Appendix A.

Distinguished and undistinguished variables are an important notion. While distinguished variables have to be bound to an individual, undistinguished variables need not; they might match “inferred individuals”. This distinction is formalized in detail in Definition 4. As a result, each SPARQL-DL query allows for undistinguished variables only in position of an individual while allowing for distinguished variables in all other positions⁸. To make a clear distinction between undistinguished variables and distinguished variables in a query, I prefix undistinguished variables (elements of $U(Q) \subseteq \mathcal{V}_{bnode}$) with ! sign, and distinguished variables (elements of $V(Q) \subseteq \mathcal{V}_{var}$) with ? sign.

Definition 4 (SPARQL-DL semantics) For a *SRQIQ* ontology \mathcal{K} with vocabulary $V = \langle CN, RN, IN \rangle$ and a SPARQL-DL query Q of the form $[q_1, \dots, q_n]$ as above, the semantics is defined as follows. A function $\mu : \mathcal{V}_{var} \rightarrow CN \cup RN \cup IN$ is a binding for Q if it is defined for all distinguished variables from $V(Q) = \{?v_1, \dots, ?v_N\}$, i.e. μ has the form $\mu = \{?v_1 \mapsto X_1, \dots, ?v_N \mapsto X_N\}$, where $X_l \in CN \cup RN \cup IN$. Each strict subset of μ is a partial binding for Q . An application of a (partial) binding μ to query atom q_l results in a new query atom $q'_l = \mu(q_l)$, where each variable $?v_k$ (from the domain of μ) is replaced by an individual $a_k = \mu(?v_k)$. An application of a (partial) binding μ to Q results in a query $\mu(Q) = [\mu(q_1), \dots, \mu(q_n)]$.

satisfaction A semi-ground query Q^S is satisfied by interpretation \mathcal{I} , denoted $\mathcal{I} \models Q^S$, if there exists an extension σ of \mathcal{I} , providing a mapping for undistinguished variables to domain elements that satisfies each semi-ground atom $q_i^S \in Q^S$. The satisfaction, denoted $\mathcal{I} \models_\sigma q_i^S$, is defined in Table 2.6 for each semi-ground SPARQL-DL query atom.

logical consequence A semi-ground query Q^S is a logical consequence of \mathcal{K} , denoted as $\mathcal{K} \models Q^S$, if $\mathcal{I} \models Q^S$ whenever $\mathcal{I} \models \mathcal{K}$.

solution A binding μ is a solution to Q (or also valid binding) if, when applied to all (distinguished) variables in $V(Q)$, we get a semi-ground query $Q^S = \mu(Q)$, for which $\mathcal{K} \models Q^S$.

□

Intuitively, a semi-ground query is a logical consequence of an ontology \mathcal{K} , if the graph (pattern) represented by the query “can be mapped” to each model of \mathcal{K} . For an intuition behind the notion of distinguished and undistinguished variables see Figure 2.9.

To demonstrate capabilities of SPARQL-DL, consider queries related to the LUBM benchmark dataset [69], that is a synthetic OWL benchmark ontology of universities, students, teachers and respective relations between them. These queries are used for query engine implementation benchmarking in Section 5.2.1

⁸Definition of reasonable semantics for undistinguished variables in class/property positions is an open problem – natural extrapolation of the RDF existential semantics of bnodes to “anonymous class” semantics would cause that an infinite number of complex classes/properties could match given bnode.

semi-ground q^S	$\mathcal{I} \models_{\sigma} q^S$ if $\forall x, y, z \in \Delta^{\mathcal{I}}$:
$\text{Ty}(a, C)$	$\sigma(a) \in C^{\mathcal{I}}$
$\text{PV}(a_1, R, a_2)$	$(\sigma(a_1), \sigma(a_2)) \in R^{\mathcal{I}}$
$\text{SA}(a_1, a_2)$	$\sigma(a_1) = \sigma(a_2)$
$\text{DF}(a_1, a_2)$	$\sigma(a_1) \neq \sigma(a_2)$
$\text{SCO}(C_1, C_2)$	$C_1^{\mathcal{I}} \subseteq C_2^{\mathcal{I}}$
$\text{EC}(C_1, C_2)$	$C_1^{\mathcal{I}} = C_2^{\mathcal{I}}$
$\text{DW}(C_1, C_2)$	$C_1^{\mathcal{I}} \cap C_2^{\mathcal{I}} = \emptyset$
$\text{CO}(C_1, C_2)$	$C_1^{\mathcal{I}} = \Delta^{\mathcal{I}} \setminus C_2^{\mathcal{I}}$
$\text{SPO}(R_1, R_2)$	$R_1^{\mathcal{I}} \subseteq R_2^{\mathcal{I}}$
$\text{EP}(R_1, R_2)$	$R_1^{\mathcal{I}} = R_2^{\mathcal{I}}$
$\text{IO}(R_1, R_2)$	$R_1^{\mathcal{I}} = (R_2^-)^{\mathcal{I}}$
$\text{Fun}(R)$	$\langle x, y \rangle \in R^{\mathcal{I}} \wedge \langle x, z \rangle \in R^{\mathcal{I}} \implies y = z$
$\text{IFun}(R)$	$\langle x, y \rangle \in (R^-)^{\mathcal{I}} \wedge \langle x, z \rangle \in (R^-)^{\mathcal{I}} \implies y = z$
$\text{Sym}(R)$	$\langle x, y \rangle \in (R^-)^{\mathcal{I}} \implies \langle y, x \rangle \in (R^-)^{\mathcal{I}}$
$\text{Trans}(R)$	$\langle x, y \rangle \in R^{\mathcal{I}} \wedge \langle y, z \rangle \in R^{\mathcal{I}} \implies \langle x, z \rangle \in R^{\mathcal{I}}$

Table 2.6.: Interpretation of SPARQL-DL semi-ground atoms.

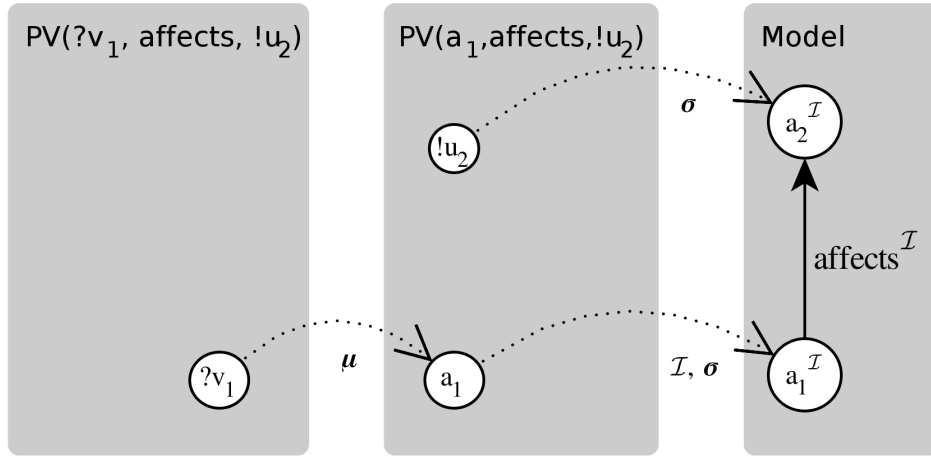


Figure 2.9.: The difference between undistinguished and distinguished variables is demonstrated on a simple SPARQL-DL query $[PV(?v_1, \text{affects}, !u_2)]$, SPARQL syntax of which is in Figure 2.4. While distinguished variables are bound to individuals (that are always interpreted as domain elements), undistinguished variables (in this case $!u_2$) might be satisfied with a domain element (here $a_2^{\mathcal{I}}$) that is not an interpretation of any individual in \mathcal{K} .

Example 7 (Distinguished and undistinguished variables) “Find all members ($?v_1$) of a research group ($!u_2$) together with courses they take ($?v_3$) and teachers ($?v_4$) of these courses ($?v_3$).”:

$$[PV(?v_1, \text{worksFor}, !u_2), Ty(!u_2, \text{ResearchGroup}), \\ PV(?v_1, \text{takesCourse}, ?v_3), PV(?v_4, \text{teacherOf}, ?v_3)]$$

This query shows the importance of undistinguished variables. When evaluating this query against the LUBM(1) dataset [69], more than 1000 result bindings for variables $?v_1, ?v_3, ?v_4$ are found.

If the undistinguished variable $!u_2$ is replaced by a distinguished one $?v_2$, the query has a different meaning – the research group is required to be materialized in the knowledge base (i.e. represented by an individual). This change, however, causes the query to return no binding as all research groups are only inferred from the axiomatization of the LUBM dataset and not explicitly represented by individuals in the ontology.

Example 8 (Variables in property position) “Find all the graduate students ($?v_1$) that are related ($?v_2$) to a course ($?v_3$) and find what kind of relationship (e.g. takesCourse , $\text{teachingAssistantOf}$) it is”:

$$[Ty(?v_1, \text{GraduateStudent}), PV(?v_1, ?v_2, ?v_3), Ty(?v_3, \text{Course})].$$

Example 9 (Querying TBox and ABox) “Find all students ($?v_1$) who are also employees and find what kind ($?v_2$) of employee (e.g. ResearchAssistant) they are”:

$$[\text{Ty} (?v_1, \text{Student}), \text{Ty} (?v_1, ?v_2), \text{SCO} (?v_2, \text{Employee})].$$

Example 10 (Querying RBox and ABox) “Find all people ($?v_1$) that teach some course ($!u_2$) and that are members of Department0. The type of their membership (worksFor, headOf) is captured by the distinguished variable $?v_3$.”

$$[\text{SPO} (?v_3, \text{memberOf}), \text{PV} (?v_1, ?v_3, \text{Department0}), \\ \text{Ty} (?v_1, \text{Person}), \text{PV} (?v_1, \text{teacherOf}, !u_2)].$$

Authors of [31] didn’t provide techniques for SPARQL-DL evaluation. I will discuss this problem in Chapter 4 and present my novel evaluation and optimization techniques for SPARQL-DL.

2.7.2. Conjunctive ABox Queries

Conjunctive ABox queries are a significant subset of SPARQL-DL queries. However, they were discussed in literature [70], [71] long time before introduction of SPARQL-DL in [31]. Contrary to SPARQL-DL, for conjunctive ABox queries there are evaluation techniques (presented in the next section) known from literature, as well as implemented in OWL reasoners.

Roughly speaking, conjunctive ABox queries ask for ontology individuals that comply with some graph pattern. As conjunctive ABox queries can be expressed by means of SPARQL-DL, I will reuse the syntax and semantics introduced in Section 2.7.1 to define them⁹.

Definition 5 (Conjunctive ABox Query) A conjunctive ABox query Q is a SPARQL-DL query

$$[q_1, \dots, q_n],$$

where each query atom q_i is of the form $\text{Ty}(a, C)$, or $\text{PV}(a_1, R, a_2)$ and $a_{(i)} \in IN \cup \mathcal{V}_{var} \cup \mathcal{V}_{bnode}$, $C \in S_C$, and $R \in RN$. Semi-ground conjunctive ABox queries are denoted as boolean queries.

□

Definition 6 (Query Graph) A graph of a conjunctive ABox query Q is a directed labeled graph $G_Q = \langle \mathcal{V}_Q, \mathcal{E}_Q, \mathcal{L}_Q \rangle$, where the set of vertices $\mathcal{V}_Q = V(Q) \cup U(Q) \cup I(Q)$ is the set of distinguished variables, undistinguished variables and individuals that occur

⁹Previous works on conjunctive ABox queries used different syntaxes. E.g. the boolean query $[\text{Ty} (!u_1, \text{GraduateStudent}), \text{PV} (!u_1, \text{takesCourse}, !u_2), \text{Ty} (!u_2, \text{Course})]$ would be rendered in [70] as $!u_1 : \text{GraduateStudent} \wedge \langle !u_1, !u_2 \rangle : \text{takesCourse} \wedge !u_2 : \text{GraduateStudent}$, and in [71] as $\text{GraduateStudent}(!u_1) \wedge \text{takesCourse}(!u_1, !u_2) \wedge \text{GraduateStudent}(!u_2)$.

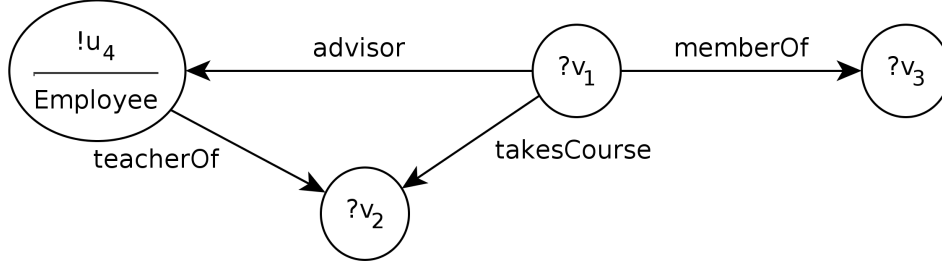


Figure 2.10.: Query graph G_{Q_1} for Q_1 .

in some $q \in Q$. The set of directed edges \mathcal{E}_Q is a set of pairs $\langle a_1, a_2 \rangle$, where atom $PV(a_1, R, a_2) \in Q$ and the labeling \mathcal{L}_Q is defined as $\mathcal{L}(a) = \{C \mid Ty(a, C) \in Q\}$ and $\mathcal{L}(a_1, a_2) = \{R \mid PV(a_1, R, a_2) \in Q\}$. Furthermore, Q has a cycle over $\mathcal{V}'_Q \subseteq \mathcal{V}_Q$ whenever the undirected subgraph $Sub(G_Q, \mathcal{V}'_Q)$ of G_Q induced by \mathcal{V}'_Q contains a cycle. A query Q has a cycle, if Q has a cycle over \mathcal{V}_Q .

□

Example 11 Let's take the following query Q_1 (with the query graph shown in Figure 2.10) into the LUBM dataset [69]. “Find all students ($?v_1$) that take courses ($?v_2$) taught by their advisors ($!u_4$). Retrieve also the courses and the affiliation ($?v_3$) of the students”:

$$[PV(?v_1, \text{advisor}, !u_4), PV(!u_4, \text{teacherOf}, ?v_2), Ty(!u_4, \text{Employee}), \\ PV(?v_1, \text{takesCourse}, ?v_2), PV(?v_1, \text{memberOf}, ?v_3)],$$

and thus $V(Q_1) = \{?v_1, ?v_2, ?v_3\}$, $U(Q_1) = \{!u_4\}$, and $I(Q_1) = \emptyset$. Although G_{Q_1} contains a cycle over $\{?v_1, ?v_2, !u_4\}$, the only undistinguished variable in the cycle is $!u_4$.

2.7.3. Evaluating Conjunctive ABox Queries

So far, significant attention has been paid to *boolean queries*, i.e. queries that test just the possibility to map a query pattern to all models of an ontology, without retrieving any variable binding. In [70] the problem of answering boolean queries is reduced to the checking of concept satisfiability for the \mathcal{ALC} language [48]. To get rid of variables, the authors use *rolling-up technique* that reformulates evaluation of a general boolean query as a set of instance retrievals or instance checks. Although for \mathcal{ALC} the proposed technique works fine, the authors noticed that its generalization beyond \mathcal{ALC} towards expressive description logics (e.g. \mathcal{SROIQ}) is problematic and it is possible only (i) for logics employing the *tree-model property* [48], or (ii) for queries, graph of which does not contain cycles made of undistinguished variables.

These limitations of boolean query answering using tableau algorithms and rolling-up technique were overcome in [64] by introducing a specialized tableau algorithm to directly check entailment of boolean queries over \mathcal{SHIQ} ontologies (a \mathcal{SROIQ} subset). The modified tableau algorithm extends the standard one for consistency checking in \mathcal{SHIQ}

[72] with a novel blocking condition¹⁰. However, this technique to be applicable requires all roles in a conjunctive query to be *simple*, i.e. not transitive and without transitive subroles. As discussed in [64], the same approach can be generalized to unions of boolean conjunctive queries in a straightforward manner.

The presence of transitive roles in query atoms is handled in [73] and [74] for description logics *SHIQ* and *SHOQ* (another subset of *SROIQ*). In both works, the respective logic (*SHIQ*, resp. *SHOQ*) is extended with *role conjunction* construct to capture “shortcuts” introduced by a transitive role. This work has shown that the problem of answering conjunctive queries for both of these subsets is decidable. However, there is still no decision procedure for conjunctive query answering in the language as expressive as *SROIQ* (or even *SHOIN*, the logic backing OWL DL).

This analysis shows, that for full *SROIQ*, decidability of conjunctive queries is maintained only when the query language is restricted. Based on this, the rest of this section presents methods, that are used in state-of-the-art *SROIQ* tableau reasoners for evaluation of conjunctive queries without cycles over undistinguished variables.

Let’s have an ontology \mathcal{K} , a *SROIQ* tableau reasoner as specified in Section 2.6, and a conjunctive query Q without cycles of undistinguished variables. Before evaluating Q against \mathcal{K} it is necessary to check consistency of \mathcal{K} . If the initial consistency check fails, the query answering procedure stops with an empty result (for non-boolean Q) or *false* (for boolean Q). If the initial consistency check succeeds, the tableau reasoner constructs the *completion* and *precompletion* structures along the way, as described in Section 2.6. Assume that \mathcal{K} is consistent.

In the rest of this thesis, I assume only conjunctive ABox queries, the graph of which is connected. Each query, a graph of which contains more connected components, can be split into several queries that can be evaluated independently for efficiency reasons and their results combined in the end, see [71].

Boolean Queries

A simple way to enforce the semantics of a boolean query Q , shown in Algorithm 1, is presented in [70]. This algorithm makes use of so called *rolling-up technique*, represented by function $\text{ROLL}(Q, x)$, that transforms a boolean query Q , for which $\text{Sub}(G_Q, U(Q))$ is a tree¹¹, into a single $C(\bullet)$ query atom. The rolling-up technique describes a query Q by a complex concept description “from the position of a given term x ”, by iteratively replacing each query atom $\text{PV}(x, R, y)$ with a query atom $(\exists R \cdot C_Y)(x)$, where C_Y is a concept that represents the rest of the query rolled-up into y in a similar way. As presented in [70], Algorithm 1 is a decision procedure for boolean queries Q for which $\text{Sub}(G_Q, U(Q))$ is a tree. On line 3 boolean queries with at least one individual are

¹⁰That preserves the blocking in a tableau algorithm to occur too early and thus ensures existence of a syntactic mapping from a boolean query to each of resulting set of completion trees if such a mapping exists.

¹¹Individuals are not considered in the subgraph, as they do not violate the applicability of the rolling-up technique: each individual \mathbf{a} can be forked into two vertices to break the cycle without any semantic impact, as $\mathbf{a}^{\mathcal{I}} = \nu(\mathbf{a})$ for any extension ν of the interpretation function \mathcal{I} .

Algorithm 1 Evaluation of Boolean Queries.

Input: Consistent \mathcal{K} ; boolean query Q .

Output: *true* if $\mathcal{K} \models Q$, *false* otherwise.

```
1: function EVALBOOL( $\mathcal{K}, Q$ )
2:   if some  $q \in Q$  references an individual  $\mathbf{a}$  then
3:     if IC( $\mathcal{K}, \text{ROLL}(Q, \mathbf{a}), \mathbf{a}$ ) then return true
4:   else
5:     if CC( $\mathcal{K} \cup \{\text{ROLL}(Q, !u) \sqsubseteq \perp\}$ ) = false for some  $!u$  then return true
6:   return false
```

evaluated by a single IC call, while line 5 handles queries with only undistinguished variables as follows: whenever $\text{ROLL}(Q, !u) \sqsubseteq \perp$ is the cause for inconsistency, it must be the case that $(\text{ROLL}(Q, !u))^{\mathcal{I}}$ is non-empty in each model \mathcal{I} of \mathcal{K} , and thus $\mathcal{I} \models Q$.

Example 12 *Let's continue with Example 11 and consider $Q_{1g} = \mu_{12}(Q_1)$ as follows ($\mu_{12} = \{?v_1 \mapsto \text{Jim}, ?v_2 \mapsto \text{Math}, ?v_3 \mapsto \text{DeptMath}\}$):*

$$\begin{aligned} & [\text{PV}(\text{Jim}, \text{advisor}, !u_4), \text{PV}(!u_4, \text{teacherOf}, \text{Math}), \text{Ty}(!u_4, \text{Employee}), \\ & \quad \text{PV}(\text{Jim}, \text{takesCourse}, \text{Math}), \text{PV}(\text{Jim}, \text{memberOf}, \text{DeptMath})], \end{aligned}$$

Q_{1g} can be evaluated by rolling-up e.g. into Jim, obtaining $\text{ROLL}(Q_{1g}, \text{Jim}) = C_{\text{Jim}}$, where

$$\begin{aligned} C_{\text{Jim}} = & \exists \text{advisor} \cdot (\text{Employee} \sqcap \exists \text{teacherOf} \cdot \{\text{Math}\}) \sqcap \exists \text{takesCourse} \cdot \{\text{Math}\} \\ & \sqcap \exists \text{memberOf} \cdot \{\text{DeptMath}\} \end{aligned}$$

and checking $\text{IC}(\mathcal{K}, C_{\text{Jim}}, \text{Jim})$, as described in Algorithm 1.

Queries without Undistinguished Variables

The authors of [71] present an algorithm for evaluating queries without undistinguished variables. To make their algorithm compliant with the notions used in this thesis, I modify their description by introducing functions NEXT and EVALATOM, resulting in Algorithm 2. Function $\text{NEXT}(\mathcal{K}, Q, \mu)$ returns a query that is a reordering of atoms in Q . As different orderings of the same set of atoms are interpreted equally (see Definition 4) a naive implementation of $\text{NEXT}(\mathcal{K}, Q, \mu)$ might return Q . Reordering of query atoms that preserves connectedness of the evaluated query, and minimizes the number of tableau reasoner runs is discussed in [71]. The method is based on a cheap preprocessing of the ontology and computing statistics (e.g. number of asserted instances of a class is used to estimate actual, inferred, number of instances of a class). Based on these statistics, for each permutation of the query atoms an execution time is estimated and the best ordering is selected for execution. Formalization of this technique and its extension towards SPARQL-DL^{NOT} queries will be presented in Section 4.2.2. Given the current binding μ , the function $\text{EVALATOM}(\mathcal{K}, q, \mu)$, shown in Algorithm 3, finds all (partial) bindings

Algorithm 2 Evaluation of conjunctive ABox queries without undistinguished variables.

Input: Consistent \mathcal{K} ; query $Q = [q_1, \dots, q_N]$; binding μ

Output: Set of all valid bindings μ' for Q .

```

1: function EVALDIST( $\mathcal{K}, Q, \mu$ )
2:   if  $Q = []$  then return  $\{\mu\}$ 
3:   else
4:      $[q, q_{p_1}, \dots, q_{p_{N-1}}] \leftarrow \text{NEXT}(\mathcal{K}, Q, \mu)$ 
5:      $\beta \leftarrow \emptyset$ 
6:     for  $\mu' \in \text{EVALATOM}(\mathcal{K}, q, \mu)$  do  $\beta \leftarrow \beta \cup \text{EVALDIST}(\mathcal{K}, [q_{p_1}, \dots, q_{p_{N-1}}], \mu')$ 
7:   return  $\beta$ 

```

$\mu' \supseteq \mu$ such that $\mathcal{K} \models \mu'(q)$. For example, $\text{EVALATOM}(\mathcal{K}, \text{PV} (?v_1, \text{teacherOf}, \text{Math}), \emptyset)$ returns a set $\{\mu_k\}$ of bindings $\mu_k = \{?v_1 \mapsto \mathbf{a}_k\}$, where $\mathbf{a}_k \in \text{IR}(\mathcal{K}, \exists \text{teacherOf} \cdot \{\text{Math}\})$.

Algorithm 2 is executed by the function call $\text{EVALDIST}(\mathcal{K}, Q, \emptyset)$ that recursively searches the state space of possible (partial) bindings and backtracks once all query atoms are evaluated. The soundness of Algorithm 2 and Algorithm 3 is presented in [71].

Algorithm 3 Evaluation of a query atom.

Input: Consistent \mathcal{K} ; query atom q ; binding μ .

Output: Set of all (partial) bindings $\mu' \supseteq \mu$ such that $\mathcal{K} \models \mu'(q)$.

```

1: function EVALATOM( $\mathcal{K}, q, \mu$ )
2:   if  $\mu(q) = \text{Ty}(\mathbf{a}, C)$  then
3:     if  $\text{IC}(\mathcal{K}, C, \mathbf{a})$  then return  $\{\mu\}$ 
4:     else return  $\emptyset$ 
5:   if  $\mu(q) = \text{PV}(\mathbf{a}_1, R, \mathbf{a}_2)$  then
6:     if  $\text{IC}(\mathcal{K}, (\exists R \cdot \{\mathbf{a}_2\}), \mathbf{a}_1)$  then return  $\{\mu\}$ 
7:     else return  $\emptyset$ 
8:    $\beta = \emptyset$ 
9:   if  $\mu(q)$  is  $\text{Ty} (?v_1, C)$  then
10:    for  $\mathbf{a} \in \text{IR}(\mathcal{K}, C)$  do  $\beta \leftarrow \beta \cup \{\mu \cup \{?v_1 \mapsto \mathbf{a}\}\}$ 
11:   else if  $\mu(q)$  is  $\text{PV} (?v_1, R, \mathbf{a}_2)$  (resp.  $\text{PV}(\mathbf{a}_2, R, ?v_1)$ ) then
12:    for  $\mathbf{a}_1 \in \text{IR}(\mathcal{K}, (\exists R \cdot \{\mathbf{a}_2\}))$ , resp.  $(\exists R^- \cdot \{\mathbf{a}_2\})$  do  $\beta \leftarrow \beta \cup \{\mu \cup \{?v_1 \mapsto \mathbf{a}_1\}\}$ 
13:   else if  $\mu(q)$  is  $\text{PV} (?v_1, R, ?v_2)$  then
14:    for  $\mathbf{a}_1 \in \text{IR}(\mathcal{K}, (\exists R \cdot \top))$  do
15:      for  $\mathbf{a}_2 \in \text{IR}(\mathcal{K}, (\exists R^- \cdot \{\mathbf{a}_1\}))$  do  $\beta \leftarrow \beta \cup \{\mu \cup \{?v_1 \mapsto \mathbf{a}_1, ?v_2 \mapsto \mathbf{a}_2\}\}$ 
16:   return  $\beta$ 

```

Handling Undistinguished Variables

In practical applications queries with both distinguished and undistinguished variables are the most common. This section describes techniques for evaluating conjunctive queries without cycles over undistinguished variables that were implemented in the Pellet reasoner, yet not published so far. Techniques described in this section try to reduce the number of calls to IC (some of which might require consistency checks). As shown in the examples below, even if obvious non-instances using completion and obvious instances using precompletion optimizations mentioned in Section 2.6 prevent other than a single (initial) consistency check to occur, the number of IC calls might be still prohibitive for an efficient evaluation.

Naive Evaluation Strategy The naive way to evaluate conjunctive ABox queries with undistinguished variables is described in Algorithm 4. On line 3, each distinguished

Algorithm 4 Naive Evaluation Strategy.

Input: Consistent \mathcal{K} ; non-boolean query Q .

Output: Set β of all valid bindings μ' for Q .

```

1: function EVALNAIVE( $\mathcal{K}, Q$ )
2:    $\beta \leftarrow \emptyset$ 
3:   for each  $\mu = \{?v_1 \mapsto a_1, \dots, ?v_N \mapsto a_N\}, ?v_k \in V(Q), a_k \in IN$  do
4:     if IC( $\mathcal{K}, \text{ROLL}(\mu(Q), a_1), a_1$ ) then  $\beta \leftarrow \beta \cup \{\mu\}$ 
5:   return  $\beta$ 

```

variable is replaced with an individual mentioned in \mathcal{K} and the resulting boolean query is evaluated using the EVALBOOL function described in Algorithm 1. The Algorithm 4 is clearly sound, as it simply tries all possible bindings. However, as shown in Example 13, the exponential blow-up is what makes it unusable for queries with more than one distinguished variable.

Example 13 *Let's evaluate Q_1 , see Example 11, against the LUBM(1) dataset. As LUBM(1) contains more than 17000 individuals the variable substitution results in about $17000^3 = 5 \times 10^{12}$ of different boolean queries to be checked on line 4 of Algorithm 4. Even if checking logical consequence of each of them might be cheap in this case (the boolean queries can be matched against precompletion and thus the only interaction with the tableau reasoner remains the initial consistency check that constructs the precompletion), Q_1 evaluation still fails to terminate within reasonable time due to the huge number of required IC operations.*

Simple Evaluation Strategy As noticed in [71], for queries with distinguished variables, the *rolling-up technique* can be used as a preprocessing step to reduce the number of boolean queries to be tested using EVALBOOL function. This approach results in Algorithm 5. For each distinguished variable $?v$ a concept $\text{ROLL}(Q, ?v)$ is computed and

instances of this concept are retrieved using the optimized IR service (line 3), as mentioned in Section 2.6. Then, each individual from the retrieved set $IN_v = IR(\mathcal{K}, \text{ROLL}(Q, ?v))$ is used as a candidate for $?v$ in the subsequent EVALBOOL calls. The soundness of this algorithm is ensured by the fact that the rolling-up technique describes only a subset of Q (as it breaks some of the cycles), but not a superset and thus cannot discard any binding that would be valid for Q .

Algorithm 5 Simple Evaluation Strategy.

Input: Consistent \mathcal{K} ; non-boolean query Q .

Output: Set β of all valid bindings μ' for Q .

```

1: function EVALSIMPLE( $\mathcal{K}, Q$ )
2:    $\beta \leftarrow \emptyset, \delta \leftarrow \emptyset$ 
3:   for  $?v \in V(Q)$  do  $\delta(?v) \leftarrow IR(\mathcal{K}, \text{ROLL}(Q, ?v))$ 
4:   for each  $\mu = \{?v_1 \mapsto \mathbf{a}_1, \dots, ?v_N \mapsto \mathbf{a}_N\}, ?v_k \in V(Q), \mathbf{a}_k \in \delta(?v_k)$  do
5:     if  $\text{IC}(\mathcal{K}, \text{ROLL}(\mu(Q), \mathbf{a}_1), \mathbf{a}_1)$  then  $\beta \leftarrow \beta \cup \{\mu\}$ 
6:   return  $\beta$ 

```

Example 14 Evaluating Q_1 against $LUBM(1)$ using Algorithm 5, Q_1 is rolled up into each distinguished variable $?v_1, ?v_2$ and $?v_3$. For $?v_3$, we get :

$$\text{ROLL}(Q_1, ?v_3) = \left(\exists \text{memberOf}^- \cdot (\exists \text{advisor} \cdot (\text{Employee} \sqcap \exists \text{teacherOf} \cdot \exists \text{takesCourse}^- \cdot \top \sqcap \exists \text{takesCourse} \cdot \top)) \right)$$

For each $?v_i$ an IR call is required. Due to the optimizations sketched in Section 2.6, the number of EVALBOOL (and thus IC) calls is significantly less than in Algorithm 4. In case of Pellet, none of these instance retrieval executions requires a consistency check and prune the number of candidates to about 3300 for $?v_1$, about 1500 for $?v_2$ and 15 for $?v_3$. Thus, the number of boolean queries, logical consequence of which is to be checked by EVALBOOL, is reduced to about $3300 \times 1500 \times 15$, i.e. 75×10^6 , still with just one (initial) consistency check.

On the other hand, each partial combination of invalid bindings is checked many times on line 5 of Algorithm 5. However, as I show in Section 4.2.1, it is not necessary to try to extend partial binding $(?v_1 \mapsto \text{Jim}, ?v_2 \mapsto \text{Math})$ with a binding for $?v_3$, whenever $\mathcal{K} \models \text{takesCourse}(\text{Jim}, \text{Math})$ does not hold. This observation could significantly reduce the number of EVALBOOL calls on line 5.

2.7.4. Distinguished Conjunctive Queries with Negation

DCQ^{NOT} were introduced in [3] to define semantics of OWL integrity constraints. DCQ^{NOT} queries cannot be expressed by means of SPARQL-DL – the construct that is missing in SPARQL-DL is *negation as failure*. Although syntax for DCQ^{NOT} used in [3] is similar to [71], I will present relevant parts from [3] in SPARQL-DL-like syntax to keep the notation compact.

Definition 7 (DCQ^{NOT}) A DCQ^{NOT} query Q (see Example 16) has the form

$$[q_1, \dots, q_M]$$

such that each atom q_i is of the form $Ty(a, C)$, or $PV(a_1, R, a_2)$, or $SA(a_1, a_2)$, or $NOT(Q_2)$, where $a_{(i)} \in IN \cup \mathcal{V}_{var}$, $C \in S_C$, $R \in RN$, and Q_2 is a DCQ^{NOT} query.

The semantics of the DCQ^{NOT} extends the semantics of SPARQL-DL in Definition 4 with interpretation of $NOT(Q_2)$ atoms – for semi-ground Q_2 with $U(Q_2) = \emptyset$, it holds that $\mathcal{I} \models_\sigma NOT(Q_2)$ whenever it does not hold that $\mathcal{I} \models_\sigma Q_2$. Since $U(Q_2) = \emptyset$, σ is equivalent to $\cdot^{\mathcal{I}}$, see Definition 4.

□

Example 15 A DCQ^{NOT} query that asks for all failures of entities not known to be Structures is as follows:

$$[Ty(?v_1, Failure), PV(?v_1, isFailureOf, ?v_2), NOT([Ty(?v_2, Structure)])]$$

Although evaluation of DCQ^{NOT} queries is not discussed in [3], an extended version of Algorithm 2 from Section 2.7.3, that supports additionally SA and NOT atoms can be used for their evaluation. This will be discussed in Section 4.2 as a part of the SPARQL-DL^{NOT} evaluation technique.

2.8. Integrity Constraints in OWL

As shown in Example 5, OWA is the fundamental principle of description logics – they are able to *infer*, but not to *validate*. However, semantic web applications and information systems need to interact with users, connect to external data sources, in which cases data validation is required. To cope with this issue, [3] proposes a special semantics for $SR\mathcal{OIQ}$ axioms, that is suitable for data validation and can be enforced by DCQ^{NOT} queries introduced in Section 2.7.4. Integrity constraints are used in Section 4.1 to express a contract between the application and the ontology.

The semantics makes possible to evaluate these axioms using CWA and the weak unique name assumption rather than the standard OWL semantics, which is based on the open world assumption and the lack of unique name assumption. Weak unique name assumption [3] ensures that two named individuals with different identities are assumed to be different (i.e. interpreted as different domain elements) unless their equality is required to satisfy the axioms in the ontology. The differences between OWL semantics and integrity constraints semantics are demonstrated in Example 16 below.

Definition 8 (Integrity Constraint) An integrity constraint β , has the syntactic form of a $SR\mathcal{OIQ}$ axiom as shown in Definition 1. β is valid w.r.t. an ontology \mathcal{K} iff there is no solution for the DCQ^{NOT} query $\mathcal{T}(\beta)$ defined in Table 2.7.

□

	β_i	$\mathcal{T}(\beta_i)$
β_1	$A_1 \sqsubseteq \forall S \cdot A_2$	$[\text{Ty} (?v_o, A_1), \text{PV} (?v_o, S, ?v_1), \text{NOT} ([\text{Ty} (?v_1, A_2)])]$
β_2	$A \sqsubseteq (\leq 1 S)$	$[\text{Ty} (?v_o, A), \text{PV} (?v_o, S, ?v_1), \text{PV} (?v_o, S, ?v_2), \text{NOT} ([\text{SA} (?v_1, ?v_2)])]$
β_3	$A \sqsubseteq (\leq n S)$	$\left[\text{Ty} (?v_o, A) \bigwedge_{1 \leq i \leq (n+1)} \text{PV} (?v_o, S, ?v_i), \bigwedge_{i < j \leq (n+1)} \text{NOT} ([\text{SA} (?v_i, ?v_j)]) \right]$
β_4	$A \sqsubseteq (\geq n S)$	$\left[\text{Ty} (?v_o, A), \right.$ $\left. \text{NOT} \left(\left[\bigwedge_{1 \leq i \leq n} \text{PV} (?v_i, S, ?v_i), \bigwedge_{i < j \leq n} \text{NOT} ([\text{SA} (?v_i, ?v_j)]) \right] \right) \right]$

Table 2.7.: Semantics of selected integrity constraints. Each construct of the form $\bigwedge_{1 \leq i \leq N} X_i$ denotes a list X_1, \dots, X_N . Definition of integrity constraints semantics for other axiom types and other class and property constructors can be found in [3].

Intuitively, each such solution to the query $\mathcal{T}(\beta)$ represents data that violate the integrity constraint β .

Example 16 Consider ontologies from Example 5, this time using integrity constraints semantics. Taking both subsumption axioms $\gamma_2 : \text{Failure} \sqsubseteq \forall \text{isFailureOf} \cdot \text{Structure}$ and $\gamma_3 : \text{Failure} \sqsubseteq (= 1 \text{ isFailureOf})$ as integrity constraints results in violation of :

- γ_3 in case of \mathcal{K}_2 (there is no object in \mathcal{K}_2 known to have failure `PillarScour`),
- γ_2 in case of \mathcal{K}_3 (`CharlesBridge` is not known to be an `Structure`), and
- γ_3 in case of \mathcal{K}_4 (there are two objects that have recorded `PillarScour` as their failure, although there must be exactly one).

E.g. γ_2 can be validated by a DCQ^{NOT} query $\tau(\gamma_2)$ from Example 15. Evaluating the query against \mathcal{K}_3 returns a solution $\mu(?v_1) = \text{PillarScour}$ and $\mu(?v_2) = \text{CharlesBridge}$, because $\mathcal{K}_3 \models \text{Failure}(\text{PillarScour})$, and $\mathcal{K}_3 \models \text{isFailureOf}(\text{PillarScour}, \text{CharlesBridge})$, but not $\mathcal{K}_3 \models \text{Structure}(\text{CharlesBridge})$.

As shown by Example 5 and 16, data validation is out of scope of traditional *SRIOQ* semantics (all ontologies are consistent in Example 5), but can be easily done using the integrity constraint semantics (integrity constraints are violated for \mathcal{K}_2 , \mathcal{K}_3 and \mathcal{K}_4). I use this feature in Section 4.1 to define a contract between the ontology and the application to ensure compatibility of the ontology with the application.

3. Ontologies in Information Systems

The idea of using semantic web ontologies as data sources for information systems has been discussed in the semantic web community for long time, see e.g. [75] for a W3C Working Group Note. This document advocates the use of semantic web ontologies in RDFS or OWL for representing domain models of OIS, that are typically designed using object-oriented principles and implemented in an object-oriented language, like Java, C# or Python. According to the authors, such ontology-backed systems should benefit from:

reuse and interoperability, as “RDF and OWL models can be shared among applications and on the web”,

flexibility, as “RDF and OWL models can operate in an open environment in which classes can be defined dynamically”,

consistency and quality checking across models,

reasoning, as “OWL has rich expressiveness supported by automated reasoning tools”.

Although these principles are clearly advantageous for information system design and operation, the process of their implementation turned out to be rather slow. Most of the problems stem from semantic incompatibility of object-oriented models and OWL ontologies – the semantic clash between OWA of semantic web ontologies and CWA of object-oriented models causes many problems in implementing the above requirements, as discussed next in this chapter.

In the next parts of this chapter, I will present a general OIS architecture proposed in [1], focusing on typical services/components that are typically provided by an OIS. Next, I will discuss different types of OIS architectures and show current approaches to their implementations together with the problems in meeting the above objectives.

3.1. Ontology-Based Information System Architecture

Authors of [1] identify widely accepted ontology life-cycle phases in ontology engineering (requirement analysis, development, integration, evaluation) and subsequent ontology usage (ontology population, cleansing, fusion, search & retrieval & reasoning). Based on this ontology evolution model, they propose a generic layered architecture of an OIS depicted in Figure 3.1. The architecture is a generalization of the NeON Toolkit [76] ontology framework, an ontology engineering environment developed within the FP6 European project NeON (IST-2005-027595). Although this framework does not provide uniform methodology for the architectural design of an OIS, it is comprehensive in terms

of services that have to be provided by an OIS – my proposal in Chapter 4 discusses *querying* and support in designing other services using an object model that reflects static parts of the ontology¹. The architecture has the following components:

presentation layer that contains sophisticated User Interface (UI) components to present ontological content,

logic layer that provides high-level ontological services for ontology engineering and ontology usage lifecycle phases

data layer that provides an abstraction over various data sources

Each particular information system might require only some components (ontology services in Figure 3.1) of the architecture. E.g. StruFail, the information system of structural failures design of which is introduced in Section 6.1, requires no components related to ontology engineering. Instead, it uses heavily “Core Ontology Services” as well as “Reasoning Related Services”, or “Ontology-based Applications’ Front-end”. On the other hand, ontology editors, like Protégé, use heavily ontology engineering services and typically not ontology usage services.

According to the authors of [1], the architecture might be instantiated in different ways, ranging from all-in-one rich clients, over tightly coupled component-oriented multi-tier Java EE architecture to loosely coupled Service Oriented Architecture (SOA). The multi-tier Java EE architecture is then demonstrated on a web-based information system use-case based on the NeON Toolkit.

OIS architectures can be roughly classified as *generic architectures* (i.e. ontology-independent architectures) and *domain-specific architectures* (i.e. ontology-dependent architectures). This distinction significantly influences how the OIS will respond to ontology changes. While systems with generic architecture do not depend on the ontology under consideration and thus, ontology changes do not influence their operation, systems with domain-specific architecture depend on (some part of) the ontology and thus, an ontology change during OIS runtime might cause runtime crash of the system. This problem will be discussed next in this section.

Although this thesis is primarily focused on OWL, resp. OWL 2, I will still address existing approaches of accessing RDFS ontologies from OISs, as there is more work done in this field, and many approaches for accessing OWL ontologies were inspired by the RDFS ones.

3.1.1. Systems with Generic Architecture

Systems introduced in this section can have different use cases, but their architecture is independent on a particular ontology *in compile-time* – their architecture (i.e. presenta-

¹Other of my results related to this thesis – OWL comparison scenario and error explanation optimization techniques that were implemented in the OWLDiff tool, contribute to *collaborative editing* and *debugging* services. To keep this thesis compact, they are not included in the main course of this thesis, but only sketched in Appendix C.

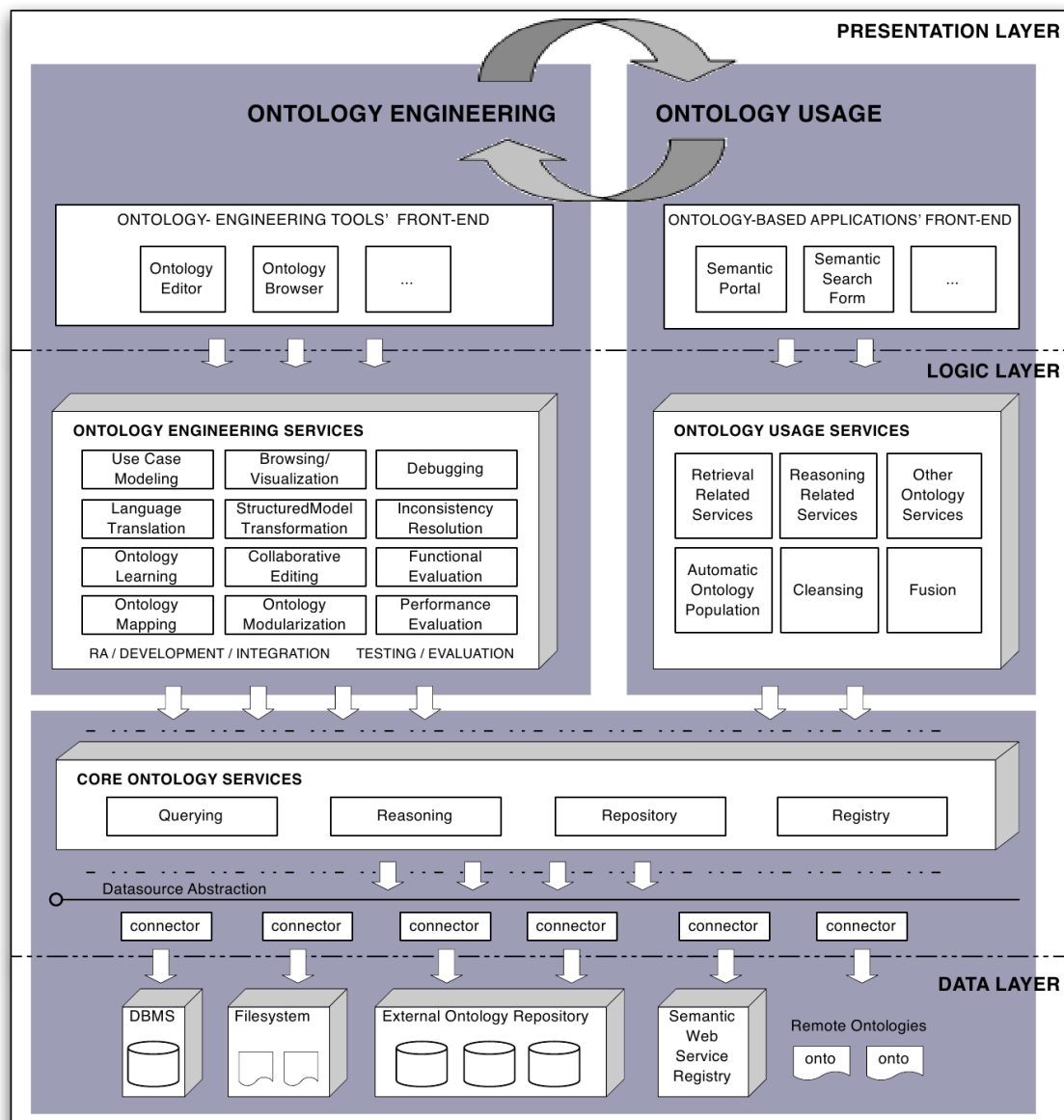


Figure 3.1.: Ontology-based system architecture as presented in [1]. (The Figure is reproduced with the permission of its authors.)

tion layer, logic layer and data layer in terms of Figure 3.1) is generic enough to access arbitrary ontologies (in the given ontology language, i.e. RDFS/OWL in our case).

ontology editors that are aimed at creating/editing/exploring ontologies. Examples of OWL ontology editors are Protégé [19], NeON Toolkit [76], TopBraid Composer [77], SWOOP [78], and other, see [79], [80], [9] or Wikipedia page “Ontology editor” for an up-to-date list.

purpose-specific systems that are tailored to specific user scenarios, comparing to ontology editors, but are still independent of the particular ontology. Examples of such scenarios are semantic search (e.g. SWOOGLE [81], an indexing and retrieval system for semantic web documents), semantic annotations of web resources (various browser plug-ins, e.g. SemanticTurkey [82] for Mozilla Firefox [83]) or semantic wikis (e.g. IkeWiki [84]).

ontology-specific systems, business components and user interface logic of which is fully described in ontologies, following the principles of Model-Driven Architecture (MDA) [85]. The business and UI logic is then generated ([86], [87]) in runtime based on the ontological descriptions, see e.g. [88], [89].

3.1.2. Systems with Domain-Specific Architecture

Architecture of domain-specific systems depends on the structure of an ontology *in compile-time*. These systems are not fully ontology-driven as they need to provide complex (procedural) business logic (e.g. complex computations) or tailored user-interfaces for the particular ontological domain of interest, that would be difficult or even impossible to express by means of semantic web ontologies. The business and UI logic is then represented by object model and its procedural logic, and the dependency of the information system on the ontology is materialized by an Object-Ontology Mapping (OOM). As shown in the next section, this dependency is usually ad-hoc which causes problems whenever ontology changes during information system runtime.

Most of the ontology-based information systems fall into this group – this can be also observed by the number of differed OOM techniques presented in Section 3.2.2. Examples of these systems include a health information system [90] (UI logic is implemented in Microsoft Access forms that reflect ontology classes and properties), a laptop e-shop [91] (generated PHP scripts based on ontology classes and properties) or Nepomuk semantic desktop [92] (generated Java object model for the Task Ontology², using RDFReactor [93]).

3.2. Accessing OWL Ontologies Programmatically

There were designed lots of approaches for implementation of the architectures presented in the previous section. Although there are some approaches for programmatic access

²See <http://www.semanticdesktop.org/ontologies>, cit.10/12/2011.

to ontologies in other languages (see [94] for Perl, [95] for Python, [96] for PHP, or ActiveRDF [24] for Ruby), most of the current approaches are tailored to Java, due to its success in the semantic web community.

A good overview of the existing approaches is presented in [97]. The approaches can be classified as (i) low-level (type 1) or (ii) high-level (type 2), according to the type of the abstraction level of their object-model – type 1 approaches are generic, their object model is independent on the particular ontology, but the programmatic use is rather verbose. On the other hand, type 2 approaches map (part of) the ontology into the object model, which makes the resulting object model dependent on the particular ontology and application access more compact. An example of this, taken from [2], is presented in Figure 3.2.

3.2.1. Type 1 APIs

As representatives of type 1 APIs for accessing OWL ontologies from Java consider two wide-spread open-source libraries – OWLAPI [57] and Jena [58].

OWLAPI (originally named WonderWeb API) has been developed by the University of Manchester for about 10 years. The 2.x versions of the API were aimed at providing access to OWL ontologies. Since 2007, its developers started to implement OWL 2 features. Finally, OWLAPI 3 became one of the reference implementations of the set of OWL 2 W3C recommendations released in 2009. It provides (i) OWL 2 metamodel that is a direct implementation of the functional syntax standardized in [18], (ii) API for basic reasoning services, e.g. consistency checking, or query answering, and (iii) API for advanced ontology services, e.g. axiom justifications, or OWL storage back-ends. Nowadays, OWLAPI is taken as a standard API for accessing OWL and OWL 2 ontologies from Java, claiming more than 20000 downloads for the last two years³.

Jena has been developed by the HP Labs Semantic Web Programme for about 10 years. Although aiming initially on the programmatic access to RDF, RDFS and OWL ontologies from Java, Jena is currently much broader project covering SPARQL processing (ARQ), storage mechanisms (TDB, SDB), and other associated projects. However, the current version of Jena (2.6.4) still lacks full OWL 2 support. Still, Jena remains an important stake-holder in this field, claiming nearly 95000 downloads for the last two years⁴.

Using APIs of type 1 is useful for developing generic systems described in Section 3.1.1. However, their use for development of domain-specific applications is typically time consuming and error-prone, as the application developer is forced to produce lots of

³see <https://sourceforge.net/projects/owlapi/files/stats/timeline?dates=2009-08-11+to+2011-08-11>, cit. 11/8/2011

⁴see <https://sourceforge.net/projects/jena/files/stats/timeline?dates=2009-08-11+to+2011-08-11>, cit. 11/8/2011

```

Collection<Person> people = a_factory.getMembers(Person.class);
for (Person person : people) {
    for (String emailAddress : person.getEmailAddress()) {
        if ("FullProfessor7@Department0.University0.edu".equals(emailAddress)
            ) {
            System.out.printf("Name:␣%s", person.getName());
        }
    }
}
}

```

```

static final String U_URI = "http://www.lehigh.edu/zhp2/2004/0401/univ-
    bench.owl#";
Literal emailAddress = ResourceFactory.createPlainLiteral("
    FullProfessor7@Department0.University0.edu");
OntClass personClass = model.getOntClass(U_URI + "Person");
OntProperty nameProperty = model.getOntProperty(U_URI + "name");
OntProperty emailProperty = model.getOntProperty(U_URI + "emailAddress")
    ;
ExtendedIterator<Individual> instances = model.listIndividuals(
    personClass);
while (instances.hasNext()) {
    Individual resource = instances.next();
    if (resource.hasProperty(emailProperty, emailAddress)) {
        RDFNode nameValue = resource.getPropertyValue(nameProperty);
        Literal nameLiteral = (Literal) nameValue.as(Literal.class);
        String name = nameLiteral.getLexicalForm();
        System.out.printf("Name:␣%s", name);
    }
}
}

```

Figure 3.2.: Type 1 APIs comparing to type 2 APIs. Top: Java code generated by Sapphire (type 2). Bottom: The same logic implemented in Jena (type 1) API. Jena code is much less readable (and thus maintainable) than the Sapphire code. Both examples are taken from [2].

similar source code snippets for the domain dependent logic. The verbosity of low level APIs (see Figure 3.2 bottom for a Jena example) also causes the resulting applications source code to be complex and hardly maintainable.

3.2.2. Type 2 APIs

For domain-specific architectures the verbosity of type 1 APIs was the main driving force for the semantic web community to propose ad-hoc mappings between an object model and an OWL ontology, often without properly understanding semantics of their modeling choices. Implementations of these mappings are typically able to generate type 2 APIs, which are easy to use by application developers, although they necessarily provide simplified view of the ontology.

APIs for RDF like Sommer [98], Winter [99], Elmo [100], RDFReactor [93], or RDF2Java [101] use OOM to map RDFS classes to Java classes and RDF properties to Java properties (fields with setters/getters) according to *an ad-hoc mapping*. They provide stateless proxy objects (e.g. RDFReactor) or stateful data access objects (e.g. Sommer). Most tools provide an object model generator from an RDFS ontology in compile-time. Querying capabilities of these APIs are rather limited. Sommer, RDFReactor and RDF2Java support only queries available through the generated API, e.g. “retrieve the author of a book”⁵ could be written in Sommer object model as

```
Person author = book.getAuthor();
```

Elmo, on the other hand, can be used as a part of the Sesame API [102], that in turn can be queried through a SPARQL [45] endpoint. Similarly, Winter (an extension of Sommer), supports SPARQL queries through Java annotations directly on the classes/their fields in the object model.

A case study, presented in [103], discusses and compares four Java-based approaches – Sommer, Elmo, RDFReactor, RDFJava – and a python-based one – SuRF. Authors interviewed the developers of these libraries and asked them questions regarding the key idea, use cases, necessary configuration, etc. Based on their response, the authors formulated a OOM (called *object-triple mapping* by the authors) that is generic enough to describe an ontology in the object model expressed in PathLog [104].

Empire [105], contrary to the above mentioned systems, is the first (partial) implementation of the Java Persistence API (JPA) 2.0 [106] for RDFS ontologies. It extends the set of JPA 2.0 annotations with custom annotations representing the mapping between a Java class and an RDF class. This allows software designers to make smooth transition from a database-backed JPA 2.0 code to an ontology-backed one. Furthermore, inspired by JPQL[106] queries for JPA 2.0, it is possible to issue SPARQL queries the underlying ontology through Empire.

⁵where book is defined earlier in the source code

APIs for OWL like Owl2Java [107], Sapphire [2], JAOb [108], or Jastor [109] try to provide ad-hoc mappings of OWL ontologies into object models. These systems differ in the level of approximation of the OWL (open-world) semantics in (closed-world) object model. E.g., the last mentioned approach, Jastor, stems from the original technique presented in [110]. Its authors propose a Java object model generator that takes an OWL ontology and translates it to a set of Java interfaces and classes that are interconnected to adapters, or cardinality constraints validators, with the aim to preserve as much OWL semantics as possible.

Neither of the above APIs provide querying capabilities going beyond the simple API calls, as discussed in the previous paragraph. This means, complex queries have to be simulated using multiple calls to the generated object model API (similarly to Figure 3.2) which introduces a performance overhead, as demonstrated in [2].

While most of the type 2 approaches propose ad-hoc mappings between ontologies and object models, authors of [111] introduce a model-driven architecture (MDA) to generate ontology APIs. The application designer develops a domain model of the ontology for his/her OIS using Unified Modeling Language (UML) and then transforms the domain model into the specific object-oriented programming language.

As the expressiveness of OWL is richer than the expressiveness of both object models and UML, it is obvious that the above mentioned mappings can not be lossless. Furthermore, queries, as expressive as SPARQL-DL introduced in Section 2.7.1, cannot be formulated in the existing APIs. All the same, existing ontology mappings and proposed APIs are useful for rapid prototyping of applications, see e.g. [112].

Statically vs. Dynamically Typed Languages

An interesting problem relevant to type 2 APIs has been posed in [75] and in [24]. The authors identify mismatches between RDFS ontologies and statically-typed⁶ object-oriented languages:

Class membership – an object cannot change its type during runtime in statically typed object-oriented languages (e.g. Java), contrary to RDFS ontologies, in which an individual can change its class membership.

Inheritance – multiple-inheritance is typically not possible in statically typed languages, contrary to RDFS.

Object conformance and semi-structured data – objects in statically typed languages have fixed structure (fields, methods), while two RDFS individuals belonging to the same class can have different properties.

Runtime Evolution – both ontological schema and ontological data can evolve at runtime, contrary to object model, that is static.

⁶Examples of statically typed languages are Java, or C#, contrary to Smalltalk, Perl, Python or Ruby that are dynamically typed, i.e. the type/class of an object can change during runtime.

The authors of [24] argue that these mismatches make statically typed languages inconvenient for accessing ontological data and propose using dynamically typed “scripting” languages, e.g. Ruby. This argument is objected in [103] by claiming that ontology schema tends to be static in practical applications. My experience is that the truth is in the middle: some parts of the schema established by domain experts are rather static, while other parts are being changed by domain experts on the fly. A significant disadvantage of dynamic typing is its inconvenience for data validation, which is often required for user inputs. My solution to this problem, presented in Chapter 4, is to split the schema into the static and dynamic parts within a statically typed object-oriented language, and thus exploit benefits of both approaches.

3.3. Relationship of this Thesis to Related Work

This section wraps Chapter 2 and Chapter 3 and relates them to the contributions of this thesis. This thesis presents a framework and a methodology for building information systems based on evolving ontologies, providing an expressive query language with efficient evaluation and optimization techniques and object-oriented API conforming to the ontology-application contract, together with prototypical implementations.

3.3.1. Ensuring Proper Application-Ontology Contract

One significant drawback of the current object-ontology mappings approaches introduced in Section 3.2.2 is that they do not take into account potential ontology evolution (see discussion to Figure 3.1 above) during the life of the information system. Hence, once a mapping is established, it reflects the current snapshot of the ontology, but no indicator exists to detect that the domain object model gets obsolete due to ontology evolution.

In contrary to these “ad-hoc” approaches described above, my approach, introduced in Section 4.1, makes the contract between the information system and the ontology explicit and formal. The proposed contract guards compatibility of the application with the ontology and prevents application crash caused by unexpected data in the ontology. Also, the contract can avoid unintentional putting of incomplete or inconsistent knowledge to the ontology by the application. From the point of object-ontological mapping, my approach lies in the middle between the type 1 and type 2 approaches to programmatic access to ontologies, described in Section 3.2. The reason is that only relevant parts of the ontology are bound to the application by the formal contract (similar to type 2 approaches introduced in Section 3.2.2), while the rest of the ontology is accessed in a generic manner by the application (similarly to the type 1 approaches introduced in Section 3.2.1).

Furthermore, the introduced approach allows to clearly distinguish competences in the application development process. Ontology and formal contract maintenance is the responsibility of the knowledge engineer educated in ontologies, but not necessarily having sufficient programming skills. On the other hand, the application developer(s) needn’t to be educated in ontological reasoning and knowledge engineering. Hence, this

approach has the potential to reduce the application development costs.

3.3.2. Providing Expressive Query Language with Efficient Implementation

Expressive query language and optimized evaluation techniques are important parts of an OISs, as also demonstrated on StruFail queries in Section 6.1. Based on the analysis introduced in Section 2.7, I introduce SPARQL-DL^{NOT}, an extension of SPARQL-DL with additional OWL 2 constructs and negation as failure, a feature that allows to use the query language not only for query answering but also for integrity constraints checking [3]. Significant contribution of this thesis includes the first SPARQL-DL and SPARQL-DL^{NOT} evaluation technique and with their implementations in the leading OWL 2 reasoner Pellet and OWL2Query engine respectively. The evaluation techniques makes use of a simple yet well-defined API to existing OWL 2 reasoners presented in Section 2.6. Furthermore, I designed, implemented and tested various optimizations of the query engine, that are described in Section 4.2 :

- *dynamic query reordering* that is useful for long queries (containing 8 or more query atoms) for which the exponential blow-up of exploring all possible static reorderings is prohibitive,
- *down-monotonic variables* making use of cheap taxonomy computation to prune invalid variable bindings,
- *splitting the query into cores* that significantly decreases execution times when both distinguished and undistinguished variables (see Definition 5) are present. Comparing to [73], [74], and [64], my optimizations are applicable to conjunctive queries without cycles through undistinguished variables for a wide range of description logics, including *SHOIN* and *SROIQ*.

4. Proposed Methodology and Framework

This chapter introduces my proposal of a framework for designing ontology-backed information systems. The framework combines an object-ontology mapping with precise definition of the contract semantics together with expressive ontological queries.

4.1. Ontology Persistence Layer

By their nature, ontologies accept *open world assumption* to capture incomplete and quickly changing knowledge present on the semantic web, as shown in Example 5 in Section 2.4. Thus, the life-cycle of an ontology is fast to reflect evolution of the domain knowledge (e.g. in the construction engineering domain: new classes/properties of construction materials, construction elements, etc.). On the other hand, an object model of an application is closed by its nature. It is static, or slowly evolving as new versions of the application are released. As a result, a mapping established between an application object model and an ontology (e.g. OOM introduced in Section 3.2.2) tends to get deprecated time to time during the life of the application.

Having this pitfall in mind, I propose a framework for accessing ontologies from applications that should consist of the following parts: (i) a contract between the ontology and the application, (ii) an object model that represents this contract in the target object-oriented language, (iii) a platform-specific control logic that ensures transactional ontology access and expressive querying capabilities, with the following requirements:

- **contract stability** – the contract has to be static or slowly evolving comparing to the ontology; the interface shall survive most ontology refinements,
- **contract maintainability** – the contract between an ontology and the respective object model has to be easy to establish and maintain,
- **non-restrictive** – the framework has to provide full access to the ontological knowledge, including entailment checking and expressive query answering,
- **validation** – the framework has to ensure that modification of the ontology by the application violates neither the consistency of the ontology, nor the contract between the application and the ontology.

Integrity constraints introduced in Section 2.8 fit well the needs of a contract between an application and an ontology, as (i) they have well-defined semantics that can be checked using OWL 2 DL reasoners that, in addition to tableau consistency checking, support also DCQ^{NOT} queries introduced in Section 2.7.4. Examples of these implementations are Pellet conjunctive query engine [32], or RacerPro nRQL engine [113], (ii) their semantics adopts CWA and thus corresponds well to the close world character of the object model, and (iii) they reuse OWL syntax which makes them *easy to author and maintain* using standard OWL ontology editors.

Reusing OWL syntax for different semantics is subject of discussion in the community. Experience proves that in this case, common syntax to integrity constraints and ontological axioms is an advantage as it significantly simplifies integrity constraint authoring and management using state-of-the-art OWL tools. For example in Protégé, the integrity constraint designer creates a new OWL document for storing integrity constraints for the developed application. Then, using import mechanism of OWL the original ontology is imported and new integrity constraints could be easily constructed based on the vocabulary and ontological axioms of the original ontology. In the proposed methodology, integrity constraints are stored in a different OWL document than ontological axioms and, additionally, are distinguished using OWL annotations (standardized in OWL 2), see Figure 5.2 in Section 5.1.

To ensure *stability* of the interface, the contract should be designed with as few integrity constraints as possible for proper application functionality. The more integrity constraints the contract contains, the less stable the contract is, resulting in more frequent object model revision.

4.1.1. Ontology-Object Model Contract

As discussed above, the contract between an ontology and an application is expressed by means of integrity constraints [3]. As integrity constraints are more expressive than object models, only some integrity constraint types can be directly expressed by means of object model constructs. Let's consider a *SROIQ* ontology \mathcal{K} , a set \mathcal{K}_C of integrity constraints, a set $CN(\mathcal{K}_C)$ (resp. $RN(\mathcal{K}_C)$) of all named concepts (resp. named roles) that occur in some integrity constraint $\gamma \in \mathcal{K}_C$. To define the translation between the integrity constraints and the corresponding object model, I introduce the mapping L that assigns¹

- each named concept $A \in CN(\mathcal{K}_C)$ an object class $L(A)$, and
- each named role $R \in RN(\mathcal{K}_C)$ a data field $L(R)$,

in the specified object programming language L . Furthermore, I denote $L(A, a)$ a run-time object (instance) of the object class $L(A)$, for which $\mathcal{K} \models A(a)$.

Basically, three types of integrity constraints with regard to their relation to the object model and their evaluation strategy can be distinguished:

¹Do not mix object language L with completion graph labeling function L_G used in Section 2.5.1.

- **compile-time** constraints are those that can be compiled into the object model of the object-oriented language under consideration. These constraints have the form β_1 and β_2 in Table 2.7 and are easy to validate, as the validity check is performed directly by the OO programming language compiler/interpreter/run-time and respective DCQ^{NOT} queries do not need to be evaluated. Integrity constraints of these types restrict the type of data field $L(S)$ within a class $L(A)$:

1. whenever both β_1 and β_2 are present, the data field $L(S)$ is of type $L(A_2)$,
2. whenever only β_1 is present, the type of $L(S)$ is a set of $L(A_2)$,
3. whenever only β_2 is present, the type of $L(S)$ is $L(T)$.

Example 17 *Whenever β_1 is present, the object-oriented language run-time ensures that only instances of $L(A_2)$ can be assigned to the data field $L(S)$ of an instance $L(A_1, a)$. This ensures the ontology to contain only S fillers² of type A_2 for the instance a of type A_1 . The respective DCQ^{NOT} query $\mathcal{T}(\beta_1)$ would return a non-empty result set whenever $\mathcal{K} \models A_1(a)$, $\mathcal{K} \models S(a, a_2)$, and it does not hold that $\mathcal{K} \models A_2(a_2)$ for some individual a_2 . Obviously, there cannot be any such a_2 in our example, such that $\mu = \{?v_0 \rightarrow a, ?v_1 \rightarrow a_2\}$ is a solution to $\mathcal{T}(\beta_1)$ – hence the integrity constraint is satisfied.*

- **run-time** constraints can not be compiled directly into the object model, but their validation can be optimized in run-time by cheap procedural pre-checks within the object model that save some of the full validity checks using a DCQ^{NOT} query (see Table 2.7 in Section 2.8). Currently considered run-time constraints have the form of cardinality constraints β_3 and β_4 from Table 2.7, but future extensions to other integrity constraint types is anticipated.

Example 18 *Whenever β_3 is present (see Table 2.7), the ontology access layer can check whether the number of fillers of field $L(S)$ of an instance $L(A_1, a)$ is smaller than n . If so, no a_{n+1} S filler of a exists that might bind the variable $?v_{n+1}$ such that $\mu = \{?v_0 \rightarrow a, ?v_1 \rightarrow a_1, \dots, ?v_{n+1} \rightarrow a_{n+1}\}$ is a solution to $\mathcal{T}(\beta_3)$ – hence the integrity constraint is satisfied.*

Similarly, whenever β_4 is present, the ontology access layer checks whether the number of fillers of field $L(S)$ in class $L(A)$ is smaller than n . If so, the integrity constraint is violated.

- **reasoning-time** constraints are all other integrity constraints that cannot be reduced³ to integrity constraints of above mentioned types. Reasoning-time integrity constraints cannot be cheaply checked by the ontology persistence layer itself, but have to be evaluated using the DCQ^{not} query engine.

² S filler of an individual a is an individual a_x for which $\mathcal{K} \models S(a, a_x)$

³Notice, that some integrity constraints, although not listed in Table 2.7 explicitly, can be reduced preserving their semantics into one or more constraints of type β_1, \dots, β_4 . E.g. $A_1 \sqsubseteq (= 1 S)$ can be replaced with two integrity constraints: (i) a compile-time constraint $A_1 \sqsubseteq (\geq 1 S)$ and (ii) a run-time constraint $A_1 \sqsubseteq (\leq 1 S)$.

As stated by Theorem 1 in [3], two different trade-offs of integrity constraints expressiveness and ontology expressiveness are possible : (i) whenever the ontology is of full *SRIOQ* expressiveness (full OWL 2 DL without data types), the integrity constraints must be of at most *SROI*, i.e. without cardinality restrictions, or (ii) whenever the ontology is at most *SRI*, integrity constraints might be of *SRIOQ* expressiveness. In the former case, integrity constraints that can be used to define ontology-application contract would be rather limited, supporting only constraints of the form β_1 and β_4 for $n = 1$ (in such a case the constraint β_4 could be rewritten as $A \sqsubseteq (\exists S \cdot \top)$). Integrity constraints of the form β_2 and β_3 cannot be used here as they contain cardinality restrictions. In the latter case, the ontology cannot contain cardinality restrictions and nominals (*SRI* expressiveness), but the ontology contract can benefit from all introduced types of integrity constraints $\beta_1 - \beta_4$ (*SRIOQ* expressiveness).

Note that pre-checking run-time cardinality constraints by simple counting of individuals introduced above fails whenever two or more individuals could be inferred to be the same, i.e. $\mathcal{K} \models \{a_1\} \sqsubseteq \{a_2\}$ for individuals a_1 and a_2 , (in addition to asserted $\{a_1\} \sqsubseteq \{a_2\}$ or $\{a_2\} \sqsubseteq \{a_1\}$ axioms). Fortunately, run-time cardinality restrictions are allowed in integrity constraints only in option (ii) above, which enforces the ontology to be at most *SRI*. In this case, no individuals can be inferred to be the same (as the *Merge* operation in the *SRIOQ* tableau algorithm [49] is not applicable), except individuals in the transitive closure of the \sqsubseteq relation (i.e. assertions of type $\{a_1\} \sqsubseteq \{a_2\}$). This transitive closure is maintained by the ontology persistence layer and used to distinguish same individuals during cardinality restrictions checking.

4.1.2. Ontology Access Layer

The ontology access layer executes ontological queries/updates requested by the application logic. The main task of the layer is to provide transactional access to the ontology, with respect to the ACID (**A**tomicity, **C**onsistency, **I**solation, **D**urability) requirements [114]. Atomicity, consistency and isolation is ensured by the transaction processing mechanism explained later in this section. Durability is ensured by creating a simple transaction log. As all object model changes can be expressed in terms of (i) axiom additions, or (ii) axiom removals, the transaction log can be just a list of change records of these two change types. Research of advanced aspects of durability is left to the next work.

The overall operation cycle of a single transaction is shown in the UML activity diagram in Figure 4.1. Each transaction can be seen in terms of two layers, (i) a front-end layer (left box in the figure), that manages a cache of objects of the object model and a (ii) back-end layer (right box in the figure) that serves as a primary storage of the ontology and is responsible for evaluation of ontology queries, consistency checking and complete evaluation of integrity constraint satisfaction.

As follows from [3], the relationship between the original *SRIOQ* semantics of an axiom and its integrity constraints semantics is rather subtle. E.g. ontology $\mathcal{K}_1 = \{A_2 \sqsubseteq \perp\}$ is consistent, but inserting axioms $A_1(a_1), S(a_1, a_2), A_2(a_2)$ causes its inconsistency. Hence, no check of integrity constraints (e.g. β_1 introduced above) can be performed. This is

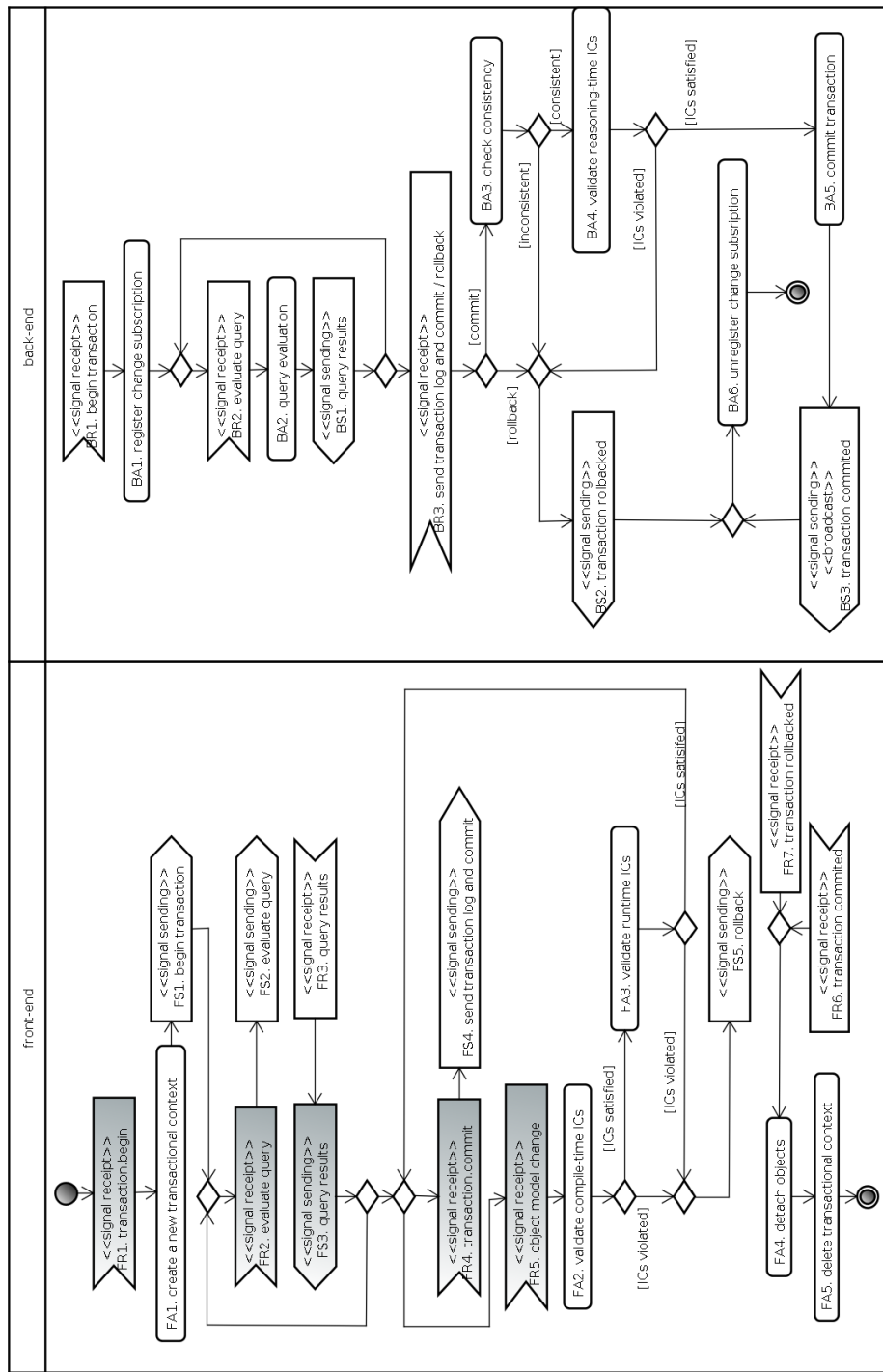


Figure 4.1.: Activity diagram of a transaction run over the Ontology Persistence layer. The business logic access to the front-end layer is represented by the shaded symbols for signal receipt/signal sending.

the reason why consistency check (BA3) has to be performed before integrity constraint violation check (BA4). On the other hand, inserting data $A_1(a_1)$, $S(a_1, a_2)$ to the ontology $\mathcal{K}_2 = \emptyset$ keeps the ontology consistent, although the integrity constraint β_1 will be violated (for a general case this integrity constraint violation is detected in BA4 in Figure 4.1 at the latest, but pre-processing in FA2 and FA3 might detect this violation in earlier stages of the life-cycle).

Since ontology consistency checking and evaluation of ontology queries is time consuming, the front-end layer optimizes integrity constraint checking (FA2 and FA3) and keeps cached the ontology changes that are propagated to the ontology at transaction commit (FS4). Due to the time and space complexity of the ontology consistency check, that is at least exponential in time, see Section 2.5.1, it is not feasible that each transaction owns a separate instance of a *SRQIQ* reasoner for its whole life duration. To handle this problem, all transactions share one instance X of a *SRQIQ* reasoner. During the commit phase, at the beginning of BA3, a new reasoner instance X' is created. When the commit succeeds (the changes violate neither integrity constraints nor ontology consistency), data is propagated to the ontology, X' becomes shared (the back-end layer benefits from serialized transactions, thus also achieving transaction isolation), replaces X (BA5) and all pending transactions are informed about reasoner change (BS3). On the other hand, transaction rollback causes X' to be disposed. The transaction scenario requires that all queries in a transaction are evaluated (FR2, FS2, BR2, BA2, BS1, FR3 and FS3) before the first data modification (FR5) in the object model to prevent reading outdated data – for the majority of practical scenarios, this restriction is not serious, as data-editing clients fetch data first, provide them to a user interface and take back changed data to be stored in the ontology.

To provide expressive query language for accessing ontological knowledge from ontology access layer, I propose the SPARQL-DL^{NOT} language introduced in the next section, evaluation of which is performed in step BA2 in Figure 4.1. The language extends both SPARQL-DL (with negation as failure and other query atom types like $\text{Ref}(P)$) and DCQ^{NOT} (with undistinguished variables and expressive query atoms). This makes it suitable for both expressive query answering and integrity constraint checking.

4.2. SPARQL-DL^{NOT} Language

Let's focus on a fundamental part of any OIS – expressive queries. In this section I present SPARQL-DL^{NOT}, my extension of SPARQL-DL introduced in Section 2.7 by novel OWL 2 DL query atoms and negation as failure. Afterwards, I present my proposal for SPARQL-DL^{NOT} evaluation and optimization. I published earlier versions of the techniques presented in this section in [29] as the first proposal for SPARQL-DL evaluation and optimization.

Definition 9 (SPARQL-DL^{NOT}) *A SPARQL-DL^{NOT} query Q extends a SPARQL-DL query in Definition 3 in the possibility of using extra query atoms:*

$$q \leftarrow \text{any SPARQL-DL query atom} \mid \text{ASym}(s) \mid \text{Ref}(s) \mid \text{IRef}(s) \mid \text{NOT}(Q_N),$$

where $s \in SN \cup \mathcal{V}_{var}$, and Q_N is a SPARQL-DL^{NOT} query without NOT atoms. Also, Q_N must not share undistinguished variables with any other atom $q \in Q$. Respective parts of Definition 3 are modified as follows:

- Set $V(Q) \subseteq \mathcal{V}_{var}$ (resp. $U(Q) \subseteq \mathcal{V}_{mode}$, resp. $I(Q) \subseteq IN$) consists of all distinguished variables (resp. undistinguished variables, resp. individuals) that appear as arguments of some query atom of Q , or in $V(Q_N)$ (resp. $U(Q_N)$, resp. $I(Q_N)$), for any atom NOT (Q_N) $\in Q$.

Semantics of semi-ground ASym, Ref, IRef, and NOT query atoms is shown in Table 4.1.

□

semi-ground q^S	$\mathcal{I} \models_{\sigma} q^S$ if $\forall x, y \in \Delta^{\mathcal{I}} :$
Ref (S)	$\langle x, x \rangle \in S^{\mathcal{I}}$
IRef (S)	$\langle x, x \rangle \notin S^{\mathcal{I}}$
ASym (S)	$\langle x, y \rangle \in S^{\mathcal{I}} \Rightarrow \langle y, x \rangle \notin S^{\mathcal{I}}$
NOT (Q_N)	there exists no binding μ' for Q_N and no extension σ' of σ , such that $\mathcal{I} \models_{\sigma'} \mu'(Q_N)$

Table 4.1.: Interpretation of extra SPARQL-DL^{NOT} semi-ground atoms, complementing the Table 2.6.

While Ref, IRef and ASym leverage the expressiveness of SPARQL-DL towards OWL 2 DL, the *negation as failure* construct represented by NOT atoms makes it possible to express non-monotonic queries (and thus also DCQ^{NOT} queries) by means of SPARQL-DL^{NOT}. On the other hand, comparing to the DCQ^{NOT} queries, SPARQL-DL^{NOT} is able to use undistinguished variables in the NOT atoms.

In Definition 9, Q_N is required not to share undistinguished variables with other atoms in Q . This restriction is posed here to make the semantics of NOT atoms simple, see last line in Table 4.1, and to make applicable the rolling-up technique, introduced in Section 2.7, for evaluation of SPARQL-DL^{NOT} queries using *SRIOQ* reasoners introduced in Section 4.2.2. Without this restriction, the extended interpretation σ of undistinguished variables from Definition 4 might contain a mapping for some undistinguished variable $!u \in U(Q_N) \cap U(Q)$ for query Q where NOT (Q_N) $\in Q$. As discussed in Section 2.7.3, the rolling-up technique describes the query as a tree, breaking any potential cycle and getting rid of all undistinguished variables⁴. Because of this, $\sigma(!u)$ might be different in each independent applications of the rolling-up technique on parts of Q sharing $!u$ (e.g. Q_N on one side and Q without NOT (Q_N) on the other side).

⁴Except the case when no distinguished variable, nor individual is present, when the last undistinguished variable is kept, see Section 2.7.3.

In Section 2.7.4, I have introduced the syntax of DCQ^{NOT} to be compatible with the syntax of SPARQL-DL NOT queries – thus Example 15 shows an example of DCQ^{NOT} /SPARQL-DL NOT query that can be used for data validation. Additional constructs in SPARQL-DL NOT can be used to formulate expressive non-monotonic queries, like the following query⁵:

Example 19 (Expressive RBox Query) *Get all asymmetric strict subproperties ($?v_1$) of the memberOf property.*

[SPO ($?v_1$, memberOf), ASym ($?v_1$), NOT (EP ($?v_1$, memberOf))].

SPARQL-DL was designed to become a SPARQL entailment regime for OWL DL, see [45]. Similarly SPARQL-DL NOT is an extension of SPARQL-DL that can be used as an OWL 2 DL entailment regime for SPARQL 1.1 (see [46] for more details on entailment regimes).

4.2.1. Optimizing Conjunctive ABox Queries with Undistinguished Variables

Before introducing the SPARQL-DL NOT evaluation technique, I will present the novel *core evaluation technique* for handling undistinguished variables in conjunctive ABox queries, that I published in [30]. The technique is used for evaluation of undistinguished variables in SPARQL-DL NOT presented in Section 4.2.2. Furthermore, it optimizes evaluation of conjunctive ABox queries, comparing to the techniques introduced in Section 2.7.3. None of the techniques presented in Section 2.7.3 takes into account partial bindings of distinguished variables during their execution. This causes the rolling-up technique to lose information about distinguished variable bindings of all but one (the variable to which the query is rolled-up) distinguished variables in the query. The novel technique presented here makes use of partial distinguished variable bindings to make the IR calls more selective and also reduces the number of IC calls.

Cores

The main idea of the core evaluation strategy introduced in this section is that parts of a query Q that contain undistinguished variables can be localized to *cores* that are evaluated separately (as special query atoms $CORE_\gamma$, see below), while the rest of the query is evaluated as a query without undistinguished variables using Algorithm 2 in Section 2.7.3. As a result, invalid partial bindings are pruned at early stages of the query processing, even before other distinguished variables and some (or all) of the undistinguished variables have to be evaluated.

⁵SPARQL-DL engine in Pellet as well as SPARQL-DL NOT engine OWL2Query, both introduced in Section 5.2, support several non-monotonic query atoms. E.g. `strictSubClassOf ($?v_1$, $?v_2$)`, asking for all pairs of subclasses that are not equivalent. This query atom has the same semantics as the SPARQL-DL NOT query `[SCO ($?v_1$, $?v_2$), NOT ([EC ($?v_1$, $?v_2$)])]`. Non-monotonic atoms just refine the possibilities for traversing concept and role hierarchies and are not elaborated here. I refer the reader to the Pellet source code available at [55] for more details.

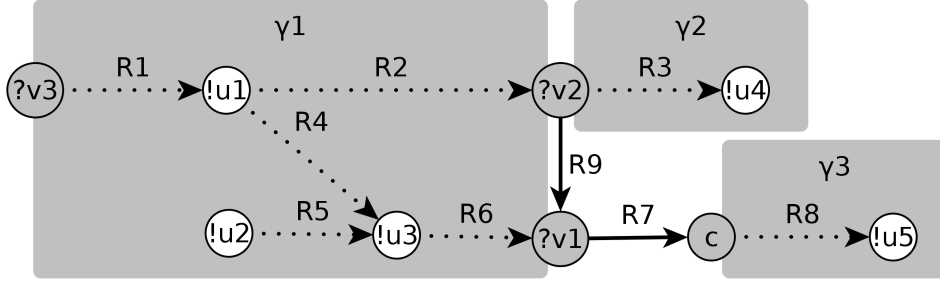


Figure 4.2.: Cores extracted from Q_X : $[PV(?v3, R1, !u1), PV(!u1, R2, ?v2), PV(?v2, R3, !u4), PV(!u1, R4, !u3), PV(!u2, R5, !u3), PV(!u3, R6, ?v1), PV(?v1, R7, c), PV(c, R8, !u5), PV(?v2, R9, ?v1)]$. Dotted arrows represent the edges of G_{Q^U} that build up the cores $\gamma_1, \gamma_2, \gamma_3$, while simple arrows represent edges of G_{Q^D} . The gray rounded rectangles demarcate cores extracted from the Q_X (maximal connected components of G_{Q^U}).

Definition 10 (Core) Consider a conjunctive ABox query $Q : [q_1, \dots, q_Z, q_{Z+1}, \dots, q_N]$ with $V(Q) \neq \emptyset$ and $U(Q) \neq \emptyset$, where no query atom from $Q^D : [q_1, \dots, q_Z]$ contains an undistinguished variable and each query atom from $Q^U : [q_{Z+1}, \dots, q_N]$ contains an undistinguished variable. A core γ is each maximal subset of Q^U , for which the graph G_γ is a maximal connected component of the graph G_{Q^U} . A signature $sig(\gamma) = V(Q) \cup I(Q)$ of γ is a set of all distinguished variables and individuals referenced in γ .

□

Note that the reordering of query atoms in Q in Definition 10 is possible, as all query atom orderings are interpreted equally, see Definition 4. Definition 10 shows how to construct cores: take all atoms from Q^U referencing any undistinguished variable from some maximal connected component of G_{Q^U} . Complex example of core partitioning is shown in Figure 4.2.

Core Evaluation

Introduction of cores makes it possible to transform each query $Q : [q_1, \dots, q_Z, q_{Z+1}, \dots, q_N]$ from Definition 10 into a new query $Q' : [q_1, \dots, q_Z, \text{CORE}_{\gamma_1}, \dots, \text{CORE}_{\gamma_K}]$ that contains no undistinguished variable ($U(Q') = \emptyset$). As demonstrated in Example 20, this transformation replaces all atoms q_{Z+1}, \dots, q_N with a set of novel atoms denoted CORE_{γ_i} , each of which represents one core (query).

Query Q' can be then evaluated with Algorithm 6. In the algorithm description on lines 6-7, newly introduced CORE_γ atoms are evaluated by calling the function `EVALCORE`, shown in Algorithm 7. The function returns all (partial) bindings $\mu' \supseteq \mu$ such that $\mathcal{K} \models \mu'(\text{CORE}_\gamma)$ iff μ' is a valid binding for γ .

Algorithm 6 Conjunctive ABox Query Evaluation using Core Evaluation Technique. The algorithm extends Algorithm 2 introduced in Section 2.7.3, with evaluation of CORE atoms using Algorithm 7.

Input: Consistent \mathcal{K} ; query $Q = [q_1, \dots, q_N]$; binding μ

Output: Set β of all valid bindings μ' for Q .

```

1: function EVALDISTCORE( $\mathcal{K}, Q, \mu$ )
2:   if  $Q = []$  then return  $\{\mu\}$ 
3:   else
4:      $[q, q_{p_1}, \dots, q_{p_{N-1}}] \leftarrow \text{NEXT}(\mathcal{K}, Q, \mu)$ 
5:      $\epsilon \leftarrow \emptyset$ 
6:     if  $q = \text{CORE}_\gamma$  then
7:        $\epsilon \leftarrow \text{EVALCORE}(\mathcal{K}, \gamma, \mu)$ 
8:     else
9:        $\epsilon \leftarrow \text{EVALATOM}(\mathcal{K}, q, \mu)$ 
10:     $\beta \leftarrow \emptyset$ 
11:    for  $\mu' \in \epsilon$  do  $\beta \leftarrow \beta \cup \text{EVALDISTCORE}(\mathcal{K}, [q_{k_1}, \dots, q_{k_{N-1}}], \mu')$ 
12:  return  $\beta$ 

```

Example 20 (Query Transformation) *Using the core evaluation technique for evaluating Q_1 from Example 11 requires its transformation into a single core γ and the transformed version Q'_1 of Q_1 :*

$$\begin{aligned}
\gamma &= [\text{Ty}(!u_4, \text{Employee}), \text{PV} (?v_1, \text{advisor}, !u_4), \text{PV} (!u_4, \text{teacherOf}, ?v_2)], \\
Q'_1 &= [\text{PV} (?v_1, \text{takesCourse}, ?v_2), \text{PV} (?v_1, \text{memberOf}, ?v_3), \text{CORE}_\gamma].
\end{aligned}$$

At this point Q'_1 is evaluated by Algorithm 6. Thus, first candidate bindings for $?v_1$, $?v_2$ and $?v_3$ are pruned iteratively using optimized IR operations (lines 12, 14, and 15 of Algorithm 3) instead of checking each candidate binding μ of variables $?v_1$, $?v_2$ and $?v_3$ using IC (line 5 of Algorithm 5). Next, when evaluating the CORE_γ atom a single IC call (as a part of the EVALBOOL call on line 4 of Algorithm 7) is required for each boolean query $\mu(\gamma)$, the number of which is significantly less than in Example 14, see Section 5.2.1.

Correctness Let's show that Algorithm 6 generates the same set of results for query

$$Q' = [q_1, \dots, q_Z, \text{CORE}_{\gamma_1}, \dots, \text{CORE}_{\gamma_K}]$$

as Algorithm 4 (and 5) for query

$$Q = [q_1, \dots, q_Z, q_{Z+1}, \dots, q_N],$$

where Q' is a transformation of Q as described in Section 4.2.1.

The following reasoning relies on the soundness of Algorithm 1 for boolean queries, Algorithm 2 for queries without undistinguished variables, Algorithm 4, and Algorithm 5

Algorithm 7 Evaluation of an Atom in a Transformed Query.

Input: Consistent \mathcal{K} ; core γ ; current binding μ .

Output: Set of all (partial) bindings $\mu' \supseteq \mu$ such that $\mathcal{K} \models \mu'(q)$, resp. $\mathcal{K} \models \mu'(\gamma)$.

```
1: function EVALCORE( $\mathcal{K}, \gamma, \mu$ )
2:    $\gamma' \leftarrow \mu(\gamma)$ 
3:   if  $V(\gamma') = \emptyset$  then
4:     if EVALBOOL( $\mathcal{K}, \gamma'$ ) then return  $\{\mu\}$ 
5:     else return  $\emptyset$ 
6:   else
7:      $\beta = \emptyset$ 
8:     for  $\mu' \in \text{EVALSIMPLE}(\mathcal{K}, \gamma')$  do
9:        $\beta \leftarrow \beta \cup \{\mu \cup \mu'\}$ 
10:  return  $\beta$ 
```

for queries with undistinguished variables. Let's take a binding μ valid for Q , found by Algorithm 4. Since $Q^D = [q_1, \dots, q_Z]$ is a subquery (without undistinguished variables) of both Q and Q' , μ is valid for Q^D and thus it will be found by Algorithm 2 for queries with distinguished variables. For each atom CORE_{γ_j} , the query γ_j , for $1 \leq j \leq K$ is constructed *only* from some atoms of $Q^U = [q_{Z+1}, \dots, q_N]$, and is evaluated with Algorithm 7, which passes the core to the function EVALBOOL on line 4, or to function EVALSIMPLE, on line 8. Thus, since μ is valid for Q , it must be also valid for γ_j .

Let's take a binding μ found by Algorithm 6. Since Algorithm 2 is sound, μ is valid for Q^D , as atoms without undistinguished variables are evaluated using the EVALATOM function. Since each atom of Q^U is present in some core γ_j , for $1 \leq j \leq K$, it must have been evaluated by function EVALBOOL, or EVALSIMPLE in function EVALCORE (Algorithm 7), when evaluating a core atom CORE_{γ_j} . Therefore, μ is valid for Q^U and thus also for Q .

□

The core evaluation tries to evaluate as many query atoms as possible using the atom-by-atom state space search for queries with distinguished variables (Algorithm 6). This brings the possibility of making use of partial bindings to prune the state space search and make IC and namely IR operations as selective as possible. The nature of Algorithm 6 allows for application of further optimizations that reflect the query shape, e.g. query reordering, described in Section 4.2.2.

4.2.2. Evaluating SPARQL-DL^{NOT}

In this section, I introduce a generic technique for evaluating SPARQL-DL^{NOT} on top of an existing *SRQIQ* reasoner, which provides a few reasoning services – IC, IR, CR, SUBC, SUPERC, ISSUBC, EQC, ISEQC, SUBP, SUPERP, ISSUBP, EQP, ISEQP – described in Table 2.5 in Section 2.6. Since the technique makes use of the core evaluation

introduced in Section 4.2.1, that in turn makes use of *rolling-up technique* introduced in Section 2.7.3, several restrictions on the use of undistinguished variables apply. Having a SPARQL-DL^{NOT} query Q ,

- no undistinguished variables can appear in DF atoms; evaluating such atoms is an open issue,
- no cycle made of undistinguished variables can appear in the query, neither in queries of the nested NOT atoms, e.g. query

$$\text{Ty} (?v, A), \text{NOT} ([\text{PV} (!u_1, R_2, !u_2), \text{PV} (!u_2, R_2, !u_3), \text{PV} (!u_3, R_3, !u_1)])$$

cannot be answered. This is a generalization of the condition already required in Section 2.7.3 for evaluating conjunctive ABox queries,

- no undistinguished variables are shared between query atoms and NOT atoms, as already required in Definition 9.

The course of the SPARQL-DL^{NOT} evaluation technique is as follows. First, a SPARQL-DL^{NOT} query Q is preprocessed resulting in a query that does not contain any SA atoms with undistinguished variables. Then, the query is evaluated using an extension of Algorithm 6, that uses the core evaluation technique to handle undistinguished variables. After introducing the evaluation technique, I introduce several optimizations, namely query reordering methods and using taxonomies to prune variable bindings for concept and role variables.

Preprocessing

Let's discuss preprocessing and normalization of SPARQL-DL^{NOT} queries. The preprocessing techniques can be divided into two groups:

required techniques that transform the query into the form suitable for evaluation using algorithms in Section 4.2.2,

optional techniques that optimize query evaluation using methods introduced in Section 4.2.2.

Techniques presented in the following paragraphs are applied for the maximal positive subquery of the original SPARQL-DL^{NOT} query, i.e. query composed of all but NOT atoms. Of course, when evaluating NOT Q atoms, the techniques are applied for Q recursively. Now, let's discuss only the required preprocessing techniques. Optional preprocessing is part of query evaluation optimizations and will be discussed in Section 4.2.2.

Required Preprocessing Techniques. As shown in Section 4.2.1, evaluation of undistinguished variables is difficult, or even impossible in general case. However, for queries without cycles of undistinguished variables an efficient technique – *core evaluation* – has been defined in Section 4.2.1, to evaluate undistinguished variables in Ty and PV atoms. At the beginning, cores of undistinguished variables are extracted from a SPARQL-DL^{NOT} query. The resulting query contains no undistinguished variables except in SA atoms (note, that as shown above no DF atoms with undistinguished variables are allowed). Thus, in the next step, such SA atoms with undistinguished variables are removed as shown in Algorithm 8, ending up with a SPARQL-DL^{NOT} query without undistinguished variables but with extra CORE atoms.

Algorithm 8 Removing SA atoms with undistinguished variables.

```

1: function SIMPLIFYSA( $Q$ )
2:    $Q \leftarrow Q \setminus \{\text{SA}(x, x)\}$ , where  $x \notin \mathcal{V}_{var}$ .
3:   if  $q = \text{SA}(!u, ?v) \in Q$ , or  $q = \text{SA}(?v, !u) \in Q$  then
4:      $\mu = \{!u \mapsto ?v\}$ 
5:      $Q \leftarrow \text{SIMPLIFYSA}(\mu(Q))$ 
6:   return  $Q$ 

```

Algorithm 8 takes a SPARQL-DL^{NOT} query Q and returns a transformed Q' such that $\mathcal{K} \models \mu(Q)$ iff $\mathcal{K} \models \mu(Q')$. On line 2, this algorithm removes all trivially satisfied atoms of the form $\text{SA}(a, a)$ from the input SPARQL-DL^{NOT} query Q . Only atoms of the form $\text{SA}(?v, ?v)$ are kept in order to produce binding for $?v$ in case no other query atom references $?v$. Next lines of the algorithm remove undistinguished variables from SA atoms by replacing each undistinguished variable by the other term in the SA atom. Let's demonstrate them on an example.

Example 21 Consider a SPARQL-DL^{NOT} query to the LUBM dataset:

$$[\text{SA}(!u_2, ?v_1), \text{Ty}(!u_2, ?v_3), \text{NOT}([\text{PV}(?v_1, \text{teacherOf}, !u_4), \text{SA}(!u_4, \text{Course1})])].$$

Note, that this query is neither conjunctive ABox query (it contains a NOT atom), neither DCQ^{NOT} query (it contains distinguished variable $?v_3$ in class position and undistinguished variables $!u_1, !u_4$). This query asks for each person $?v_1$ that is the same as another person $!u_2$, with unknown type $?v_3$ (e.g. Employee, GraduateStudent), and that is not known teacher of a course $!u_4$ that is same as Course1.

Algorithm 8 removes both SA atoms and replaces $!u_2$ with $?v_1$ and $!u_4$ with Course1, result of which is the query

$$[\text{Ty}(?v_1, ?v_3), \text{NOT}([\text{PV}(?v_1, \text{teacherOf}, \text{Course1})])].$$

The meaning of this query is the same as of the original one – the SA atoms ensured that the interpretation of undistinguished variable $!u_2$ (resp. $!u_4$) has to be the same as the interpretation of $?v_1$ (resp. Course1), which is ensured by replacing the undistinguished variables with distinguished ones in the new query.

This preprocessing cannot be used to get rid of **SA** atoms without undistinguished variables, since \mathcal{K} might contain individuals explicitly stated to be the same (e.g. $\{a_1\} \sqsubseteq \{a_2\}$ in \mathcal{SROIQ}). For example when evaluating the atom $\mathbf{SA}(?v_1, ?v_2)$, it is necessary to bind to $?v_1$ and $?v_2$ all combinations of individuals in the equivalence set induced by $\{a_1\} \sqsubseteq \{a_2\}$ axioms. Correctness of these transformations follows from the semantics of **SA** atoms in Table 2.6 and the following idea: For each interpretation \mathcal{I} , one can always choose σ as follows – $\sigma(a_1) = \sigma(\mu(a_2))$ if a_2 is a distinguished variable and $\sigma(a_1) = \sigma(a_2)$ otherwise.

Evaluating Query Atoms

After the preprocessing phase, a SPARQL-DL^{NOT} query Q contains no undistinguished variables, except in the cores γ of the **CORE** $_\gamma$ atoms. Thus, Q can be evaluated using a similar strategy as Algorithm 6. The extended algorithm is depicted as Algorithm 9. Algorithm 9 (initiated by the call $\text{EVALSDLN}(\mathcal{K}, Q, \emptyset)$) is recursive and performs a

Algorithm 9 SPARQL-DL^{NOT} Query Evaluation Procedure.

Input: Consistent \mathcal{K} ; query $Q = [q_1, \dots, q_N]$; binding μ .

Output: Set β of all solutions for Q .

```

1: function EVALSDLN( $\mathcal{K}, Q, \mu$ )
2:   if  $Q = []$  then return  $\{\mu\}$ 
3:   else
4:      $[q, q_{k_1}, \dots, q_{k_{N-1}}] \leftarrow \text{NEXT}(\mathcal{K}, Q, \mu)$ 
5:      $\epsilon \leftarrow \emptyset$ 
6:     if  $q = \mathbf{CORE}_\gamma$  then
7:        $\epsilon \leftarrow \text{EVALCORE}(\mathcal{K}, \gamma, \mu)$ 
8:     else if  $q = \mathbf{NOT}(Q_N)$  then
9:        $\epsilon \leftarrow \{\mu \cup \{?v_1 \mapsto x_1, \dots, ?v_M \mapsto x_M\} \mid ?v_i \in P, x_i \in CN \cup RN \cup IN\}$ 
10:      where  $P = V(\mu(Q \setminus \{\mathbf{NOT}(Q_N)\})) \cap V(\mu(Q_N))$ 
11:       $\epsilon \leftarrow \epsilon \setminus \text{EVALSDLN}(\mathcal{K}, Q_N, \mu)$ 
12:     else
13:        $\epsilon \leftarrow \text{EVALATOMSDLN}(\mathcal{K}, q, \mu)$ 
14:      $\beta \leftarrow \emptyset$ 
15:     for  $\mu' \in \epsilon$  do  $\beta \leftarrow \beta \cup \text{EVALSDLN}(\mathcal{K}, [q_{k_1}, \dots, q_{k_{N-1}}], \mu')$ 
16:   return  $\beta$ 
```

breadth-first search to materialize a binding for all distinguished variables in the input query Q .

Algorithm 9 evaluates **CORE** atoms on line 7, **NOT** atoms on line 11 and all other SPARQL-DL^{NOT} atoms are evaluated on line 13 by the nested call to EVALATOMSDLN described in Algorithm 10. While μ contains the variable binding constructed so far, the set ϵ contains all partial bindings μ' that extend μ and are valid for q . The particular query atoms are evaluated as follows:

- CORE_γ atoms introduced by core evaluation technique are evaluated by the original procedure EVALCORE introduced in Section 4.2.1, where its correctness is discussed. This procedure finds all solutions for a core γ that become candidate variable bindings for the next recursion step on line 15 of Algorithm 9,
- $\text{NOT}(Q_N)$ atoms are evaluated by finding all solutions for Q_N using the call to EVALSDLN . These bindings are subtracted from all possible bindings of all distinguished variables that appear in Q_N as well as Q without $\text{NOT}(Q_N)$, i.e. variables appearing in $V(\mu(Q_N)) \cap V(\mu(Q \setminus \{\text{NOT}(Q_N)\}))$. Then, the resulting bindings are passed to the next recursion step of Algorithm 9. Note, that $V(\mu(Q_N)) \setminus V(\mu(Q \setminus \{\text{NOT}(Q_N)\}))$ might be nonempty, in which case Q_N contains distinguished variables that are not contained outside the Q_N atom. These variables are not bound in the result of the query Q and thus only existence of their binding to named individuals/named concepts/named roles is checked by $\text{EVALSDLN}(\mathcal{K}, Q_N, \mu)$, to satisfy the semantic condition for NOT atoms in Table 4.1.
- other $\text{SPARQL-DL}^{\text{NOT}}$ atoms are evaluated using Algorithm 10. This algorithm evaluates $\text{SPARQL-DL}^{\text{NOT}}$ atoms in the very same manner as Algorithm 3 evaluates query atoms of conjunctive ABox queries.

Correctness of Algorithm 9, that uses Algorithm 10, follows from the correctness of Algorithm 6 and the semantics of additional $\text{SPARQL-DL}^{\text{NOT}}$ constructs. For NOT query atoms, the semantic condition $\mathcal{I} \models_\sigma \mu(\text{NOT}(Q_N))$ if there do not exist extension σ' of σ , and μ' of μ such that $\mathcal{I} \models_{\sigma'} \mu'(Q_N)$ from Section 4.2 has to be ensured. The key property is that Q_N does not share undistinguished variables with the rest of the query outside $\text{NOT}(Q_N)$. As a result, $\sigma' = \sigma \cup \sigma''$, where σ'' and σ do not overlap, whenever $\mathcal{I} \models_{\sigma''} \mu'(Q_N)$ for. The binding μ' of distinguished variables in Q_N , generated by $\text{EVALSDLN}(\mathcal{K}, Q_N, \mu)$ on line 11, exists whenever $\mathcal{I} \models_{\sigma''} \mu'(Q_N)$ and thus also $\mathcal{I} \models_{\sigma'} \mu'(Q_N)$. Thus, retracting the result of $\text{EVALSDLN}(\mathcal{K}, Q_N, \mu)$ from all possible bindings of distinguished variables in Q_N gets rid of such μ' (corresponding to such σ') in ϵ .

The correctness of the Algorithm 10 for evaluating $\text{SPARQL-DL}^{\text{NOT}}$ atoms is a straightforward extension of the correctness of Algorithm 3, as defined in [71]. If branches on lines 3, 8, as well as the first two options $\text{Ty}(a, C)$ and $\text{PV}(a_1, R, a_2)$ on line 38 are evaluated in the same manner as in Algorithm 3. Evaluation of query atoms SCO , EC , SPO , EP , Ty , PV is analogous using the corresponding operations of the reasoner interface defined in Table 2.5. Other query atoms are evaluated using the transformation of OWL entailment to SHOIN satisfiability that stems from Figure 7 of [56] and thus I will omit them here.

Let's discuss the evaluation of novel $\text{SPARQL-DL}^{\text{NOT}}$ query atoms Ref , IRef , ASym that do not follow from [56]. I will discuss only evaluation of ground atoms on line 38 and 33. Evaluation of their counterparts with distinguished variables on line 36 and 30 just

Algorithm 10 SPARQL-DL^{NOT} Atom Evaluation. The symbol $|$ denotes alternatives.

Input: Consistent \mathcal{K} ; current SPARQL-DL^{NOT} atom q' ; binding μ ; $\beta = \emptyset$.

Output: Set β of partial candidate bindings.

```

1: function EVALATOMSDLN( $\mathcal{K}, q', \mu$ )  $\triangleright$  In each iteration,  $f$  is a fresh individual not appearing in  $\mathcal{K}$ .
2:    $q \leftarrow \mu(q')$ 
3:   if  $q = \text{Ty}(?v_2, C_1) \mid \text{PV}(?v_2, R, a_1) \mid \text{PV}(a_1, R, ?v_2)$  then
4:     for  $a_2 \in \text{IR}(\mathcal{K}, C_1) \mid \exists R \cdot \{a_1\} \mid \exists R^- \cdot \{a_1\}$  do  $\beta \leftarrow \beta \cup \{\mu \cup \{?v_2 \mapsto a_2\}\}$ 
5:   else if  $q = \text{Ty}(a_1, ?v_2)$  then for  $A_2 \in \text{CR}(\mathcal{K}, a_1)$  do  $\beta \leftarrow \beta \cup \{\mu \cup \{?v_2 \mapsto A_2\}\}$ 
6:   else if  $q = \text{Ty}(?v_1, ?v_2)$  then
7:     for  $A_2 \in \text{CN}$  and  $a_1 \in \text{IR}(\mathcal{K}, A_2)$  do  $\beta \leftarrow \beta \cup \{\mu \cup \{?v_1 \mapsto a_1, ?v_2 \mapsto A_2\}\}$ 
8:   else if  $q = \text{PV}(?v_1, R_3, ?v_2)$  then
9:     for  $a_1 \in \text{IR}(\mathcal{K}, \exists R_3 \cdot \top)$  and  $a_2 \in \text{IR}(\mathcal{K}, \exists R_3^- \cdot \{a_1\})$  do  $\beta \leftarrow \beta \cup \{\mu \cup \{?v_1 \mapsto a_1, ?v_2 \mapsto a_2\}\}$ 
10:  else if  $q = \text{PV}(?v_1, ?v_3, a_2) \mid \text{PV}(a_2, ?v_3, ?v_1)$  then
11:    for  $R \in \text{RN}$  and  $a_1 \in \text{IR}(\mathcal{K}, \exists R \cdot \{a_2\}) \mid \text{IR}(\mathcal{K}, \exists R^- \cdot \{a_2\})$  do
12:       $\beta \leftarrow \beta \cup \{\mu \cup \{?v_3 \mapsto R, ?v_1 \mapsto a_1\}\}$ 
13:  else if  $q = \text{PV}(a_1, ?v_3, a_2)$  then
14:    for  $R \in \text{RN}$  do if  $\text{IC}(\exists R \cdot \{a_2\}, a_1)$  then  $\beta \leftarrow \beta \cup \{\mu \cup \{?v_3 \mapsto R\}\}$ 
15:  else if  $q = \text{PV}(?v_1, ?v_3, ?v_2)$  then for  $R_3 \in \text{RN}$  and  $a_1 \in \text{IR}(\mathcal{K}, \exists R_3 \cdot \top)$  do
16:    for  $a_2 \in \text{IR}(\mathcal{K}, \exists R_3^- \cdot \{a_1\})$  do  $\beta \leftarrow \beta \cup \{\mu \cup \{?v_1 \mapsto a_1, ?v_3 \mapsto R_3, ?v_2 \mapsto a_2\}\}$ 
17:  else if  $q = \text{SA}(?v_1, a_2) \mid \text{DF}(?v_1, a_2)$  or switched arguments then
18:    for  $a_1 \in \text{IR}(\mathcal{K}, \{a_2\}) \mid \text{IR}(\mathcal{K}, \neg\{a_2\})$  do  $\beta \leftarrow \beta \cup \{\mu \cup \{?v_1 \mapsto a_1\}\}$ 
19:  else if  $q = \text{SA}(?v_1, ?v_2) \mid \text{DF}(?v_1, ?v_2)$  then
20:    for  $a_1 \in \text{IN}$  and  $a_2 \in \text{IR}(\mathcal{K}, \{a_1\}) \mid \text{IR}(\mathcal{K}, \neg\{a_1\})$  do  $\beta \leftarrow \beta \cup \{\mu \cup \{?v_1 \mapsto a_1, ?v_2 \mapsto a_2\}\}$ 
21:  else if  $q = \text{SCO}(?v_1, C_2) \mid \text{SCO}(C_2, ?v_1)$  then
22:    for  $A_1 \in \text{SUBC}(\mathcal{K}, C_2) \mid \text{SUPERC}(\mathcal{K}, C_2)$  do  $\beta \leftarrow \beta \cup \{\mu \cup \{?v_1 \mapsto A_1\}\}$ 
23:  else if  $q = \text{EC}(?v_1, C_2) \mid \text{CO}(?v_1, C_2) \mid \text{DW}(?v_1, C_2)$  or switched arguments then
24:    for  $A_1 \in \text{EQC}(\mathcal{K}, C_2) \mid \text{EQC}(\mathcal{K}, \neg C_2) \mid \text{SUBC}(\mathcal{K}, \neg C_2)$  do  $\beta \leftarrow \beta \cup \{\mu \cup \{?v_1 \mapsto A_1\}\}$ 
25:  else if  $q = \text{SCO}(?v_1, ?v_2) \mid \text{EC}(?v_1, ?v_2) \mid \text{CO}(?v_1, ?v_2) \mid \text{DW}(?v_1, ?v_2)$  then
26:    for  $A_1 \in \text{CN}$  and  $A_2 \in \text{SUBC}(\mathcal{K}, A_1) \mid \text{EQC}(\mathcal{K}, A_1) \mid \text{EQC}(\mathcal{K}, \neg A_1) \mid \text{SUBC}(\mathcal{K}, \neg A_1)$  do
27:       $\beta \leftarrow \beta \cup \{\mu \cup \{?v_1 \mapsto A_1, ?v_2 \mapsto A_2\}\}$ 
28:  else if  $q = \text{SPO}(?v_1, R_2) \mid \text{SPO}(R_2, ?v_1)$  then
29:    for  $R_1 \in \text{SUBP}(\mathcal{K}, R_2) \mid \text{SUPERP}(\mathcal{K}, R_2)$  do  $\beta \leftarrow \beta \cup \{\mu \cup \{?v_1 \mapsto R_1\}\}$ 
30:  else if  $q = \text{EP}(?v_1, R_2) \mid \text{IO}(?v_1, R_2)$  or switched arguments then
31:    for  $R_1 \in \text{EQP}(\mathcal{K}, R_2) \mid \text{EQP}(\mathcal{K}, R_2^-)$  do
32:       $\beta \leftarrow \beta \cup \{\mu \cup \{?v_1 \mapsto R_1\}\}$ 
33:  else if  $q = \text{SPO}(?v_1, ?v_2) \mid \text{EP}(?v_1, ?v_2) \mid \text{IO}(?v_1, ?v_2)$  then
34:    for  $R_2 \in \text{RN}$  and  $R_1 \in \text{SUBP}(\mathcal{K}, R_2) \mid \text{EQP}(\mathcal{K}, R_2) \mid \text{EQP}(\mathcal{K}, R_2^-)$  do
35:       $\beta \leftarrow \beta \cup \{\mu \cup \{?v_1 \mapsto R_1, ?v_2 \mapsto R_2\}\}$ 
36:  else if  $q = (\text{Fun} \mid \text{IFun} \mid \text{Trans} \mid \text{Ref} \mid \text{IRef} \mid \text{Sym} \mid \text{ASym})(?v)$  then
37:    for  $R \in \text{RN}$  do if  $\text{ISUBC}(\mathcal{K}, \top, \leq 1 R) \mid \text{ISUBC}(\mathcal{K}, \top, \leq 1 R^-) \mid \text{ISUBC}(\mathcal{K}, \exists R \cdot \exists R \cdot \{f\}, \exists R \cdot \{f\}) \mid \text{ISUBC}(\mathcal{K}, \{f\}, \exists R \cdot \{f\}) \mid \text{ISUBC}(\mathcal{K}, \exists R \cdot \text{Self}, \perp) \mid \text{ISEQP}(\mathcal{K}, R, R^-) \mid \text{ISUBC}(\mathcal{K}, \{f\} \sqcap \exists R \cdot \neg\{f\} \sqcap \exists R \cdot \{f\}, \perp)$  then  $\beta \leftarrow \beta \cup \{\mu \cup \{?v \mapsto R\}\}$ 
38:  else if  $q = \text{Ty}(a, C) \mid \text{PV}(a_1, R, a_2) \mid \text{SA}(a_1, a_2) \mid \text{DF}(a_1, a_2) \mid \text{SCO}(C_1, C_2) \mid \text{EC}(C_1, C_2) \mid \text{SPO}(R_1, R_2) \mid \text{EP}(R_1, R_2) \mid \text{DW}(C_1, C_2) \mid \text{CO}(C_1, C_2) \mid \text{Fun}(R) \mid \text{IFun}(R) \mid \text{Ref}(R) \mid \text{IRef}(R) \mid \text{Sym}(R) \mid \text{Trans}(R) \mid \text{IO}(R_1, R_2) \mid \text{ASym}(R)$  then
39:    if  $\text{IC}(\mathcal{K}, C, a) \mid \text{IC}(\mathcal{K}, \exists R \cdot \{a_2\}, a_1) \mid \text{IC}(\mathcal{K}, \{a_2\}, a_1) \mid \text{IC}(\mathcal{K}, \neg\{a_2\}, a_1) \mid \text{ISUBC}(\mathcal{K}, C_1, C_2) \mid \text{ISEQC}(\mathcal{K}, C_1, C_2) \mid \text{ISUBP}(\mathcal{K}, R_1, R_2) \mid \text{ISEQP}(\mathcal{K}, R_1, R_2) \mid \text{ISUBC}(\mathcal{K}, C_1, \neg C_2) \mid \text{ISEQC}(\mathcal{K}, C_1, \neg C_2) \mid \text{ISUBC}(\mathcal{K}, \top, \leq 1 R) \mid \text{ISUBC}(\mathcal{K}, \top, \leq 1 R^-) \mid \text{ISUBC}(\mathcal{K}, \{f\}, \exists R \cdot \{f\}) \mid \text{ISUBC}(\mathcal{K}, \exists R \cdot \text{Self}, \perp) \mid \text{ISEQP}(\mathcal{K}, R, R^-) \mid \text{ISUBC}(\mathcal{K}, \exists R \cdot \exists R \cdot \{f\}, \exists R \cdot \{f\}) \mid \text{ISEQP}(\mathcal{K}, R_1, R_2^-) \mid \text{ISUBC}(\mathcal{K}, \{f\} \sqcap \exists R \cdot \neg\{f\} \sqcap \exists R \cdot \{f\}, \perp)$  then  $\beta \leftarrow \beta \cup \{\mu\}$ 
40:  return  $\beta$ 

```

evaluates the respective atom for each $R \in RN$ replacing⁶ the distinguished variable(s). While $\text{IRef}(R)$ atoms can be evaluated by reformulating the query with other OWL 2 constructs (see next paragraphs), evaluating $\text{Ref}(R)$ and $\text{ASym}(R)$ needs an auxiliary fresh individual f to capture cycles in the interpretation of R , similarly to the evaluation of $\text{Trans}(R)$ introduced in [56]. Details on evaluation of each of these atoms are provided next.

According to Definition 2, $\text{Ref}(R)$ is satisfied whenever all domain elements are connected to themselves by R^I in each model \mathcal{I} . This is validated by $\text{ISUBC}(\mathcal{K}, \{f\}, \exists R \cdot \{f\})$ that checks whether $\mathcal{K} \models \{f\} \sqsubseteq \exists R \cdot \{f\}$. As f is fresh individual that is not referenced elsewhere in \mathcal{K} , it is general enough to “represent” any domain element. Thus, the entailment holds whenever such R^I connection is valid for each individual f from \mathcal{K} .

According to Definition 2, $\text{IRef}(R)$ is satisfied whenever no domain element is connected to itself by R^I in each model \mathcal{I} . This is validated by $\text{ISUBC}(\mathcal{K}, \exists R \cdot \text{Self}, \perp)$ that checks that $\mathcal{K} \models \exists R \cdot \text{Self} \sqsubseteq \perp$, i.e. R^I never makes a loop over a domain element.

According to Definition 2, $\text{ASym}(R)$ is satisfied whenever no two domain elements a_1 and a_2 are connected by a relation R in both directions. This is validated by $\text{ISUBC}(\mathcal{K}, \{f\} \sqcap \exists R \cdot (\neg\{f\} \sqcap \exists R \cdot \{f\}), \perp)$ that checks that $\mathcal{K} \models \{f\} \sqcap \exists R \cdot (\neg\{f\} \sqcap \exists R \cdot \{f\}) \sqsubseteq \perp$. This condition ensures nonexistence of a domain element f^I connected by R^I to strictly another domain element (represented by a concept $\neg\{f\}$, element of which is interpreted as a different individual than f) and then back by R^I to f^I .

□

Example 22 *Evaluating the query atom $\text{Sym}(?v)$ using Algorithm 10 (line 36) tries every possible role R in RN and returns a binding $\{?v_1 \mapsto R\}$ whenever R is equivalent to its inverse, as follows from Definition 2, which is checked by $\text{ISEQP}(O, R, R^-)$ on line 37 of Algorithm 10.*

Optimizations

The syntactic overhead of SPARQL-DL makes it possible to simplify queries before their evaluation by removing (i) trivially satisfied atoms SA , SCO , EC , SPO , EP atoms (binary atoms with reflexive semantics, see Section 2.7.1) that have both arguments identical, e.g. $\text{SCO}(C_1, C_2)$; (ii) atoms with \top , \perp in particular positions, e.g. $\text{SCO}(C, \top)$, or $\text{SCO}(\perp, C)$. On the other hand, it is possible to immediately fail (i.e. return empty result for) queries containing DW or DF atoms (binary atoms with irreflexive semantics) that have both arguments identical. These preprocessing steps are not exhaustive, but they are all cheap to compute (polynomial in time comparing to exponential complexity of tableau reasoners) and since they decrease the number of atoms in the query, they are valuable especially w.r.t. the cost based reordering presented next in this section.

Also many known preprocessing techniques are applicable for SPARQL-DL^{NOT} queries. Redundant atoms can be removed using the *domain-range simplification* introduced in

⁶Note that for the sake of brevity, Algorithm 10 uses symbol R for denoting general named roles, although atoms Ref , IRef and Asym allow only simple roles. Thus, whenever speaking about these atoms in this algorithm, R is used in the meaning of a *simple role*, see Section 4.2 and 2.4.

[71]. To avoid computing Cartesian product of the query results, a SPARQL-DL^{NOT} query is split into connected components, as discussed in Section 2.7.3 for conjunctive ABox queries. Second, whenever the expressiveness of the ontology is \mathcal{ALC} , it is possible to make use of its *tree model property*, see [48], to get rid of as many undistinguished variables as possible by replacing them with distinguished ones using technique described in [72]. Last, but not least static query reordering method presented in [71] is applicable, as discussed next.

Although not primary focus of this thesis, I sketch two optimization techniques that I developed specifically for SPARQL-DL^{NOT} : (i) *static and dynamic query reordering*, and (ii) *down-monotonic variable pruning*.

Static Query Reordering. In [71] a static cost-based reordering method has been introduced for conjunctive ABox queries. The idea is to estimate the cost of evaluating a query atom by estimating the number of solutions to that query atom. The estimates are based on statistics computed by cheap preprocessing of the ontology. These estimates are then used to find a permutation of query atoms that will provide optimal execution.

For the purposes of SPARQL-DL^{NOT}, I generalized this static reordering strategy, as shown in Algorithm 11. The cost computation for given atom ordering is shown in Algorithm 12. For each query atom q two functions are needed: $\text{ESTC}(q, B)$, that estimates the cost (measured by the expected number of tableau algorithm runs) needed to evaluate an atom q , and $\text{ESTB}(q, B)$, that estimates number of execution branches generated by evaluating q (i.e. number of the partial bindings returned by Algorithm 10). Both functions take as an argument a collection B of bound variables, binding of which is unknown at the time of computing the cost.

Algorithm 11 Static Query Reordering.

Input: Consistent \mathcal{K} ; SPARQL-DL^{NOT} query Q ; set B of bound variables.

Output: Optimal reordering of Q .

```

1: function STATICNEXT( $\mathcal{K}, Q, \mu$ )
2:   if  $\mu \neq \emptyset$  then return  $Q$ 
3:    $Q^* \leftarrow Q$  and  $cost^* \leftarrow \infty$ 
4:   for  $Q_p \in \text{ALLPERMUTATIONSOFACTOMSIN}(Q)$  do
5:      $cost_p \leftarrow \text{STATICCOST}(Q_p, \emptyset)$ 
6:     if  $cost^* > cost_p$  then
7:        $cost^* \leftarrow cost_p$ 
8:        $Q^* \leftarrow Q_p$ 
9:   return  $Q^*$ 

```

Algorithm 11 executes just once, at the beginning of query evaluation (line 2 in Algorithm 11 returns the computed ordering otherwise), and “simulates” the run of the SPARQL-DL^{NOT} engine (without performing any expensive \mathcal{SROIQ} reasoning) to propose best ordering of query atoms to be executed by the real query engine. The estimates ESTC and ESTB are obtained in two different ways:

Algorithm 12 Static Ordering Cost Computation.

Input: SPARQL-DL^{NOT} query Q ; set B of bound variables

Output: Cost of the evaluation of query Q .

```
1: function STATICCOST( $Q, B$ )
2:   if  $Q = \emptyset$  then return 1
3:    $[q|Q_r] \leftarrow Q$ 
4:   return ESTC( $q, B$ ) + ESTB( $q, B$ ) · STATICCOST( $r, B \cup V([q])$ )
```

ESTC(q, B) is the estimated cost of evaluating a query atom q given that variables in B are bound. The estimated cost χ is measured in the number of tableau algorithm runs t_{CC} , as specified in Table 2.5. This estimate is computed based on the course of evaluation of q in Algorithm 10. Rough estimates that correspond to worst cases are shown in Table B.1 in Appendix B.

Example 23 *As an example, the query atom $q = \text{PV} (?v_1, ?v_2, \mathbf{a})$ is evaluated by computing all instances (instance retrieval) of a concept $\exists R \cdot \{\mathbf{a}\}$ for each $R \in RN$, thus $\text{ESTC}(q, \emptyset) = |RN| \cdot \chi(\text{IR})$, where $\chi(\text{IR}) = |IN| \cdot t_{CC}$ as defined in Table 2.5 in Section 2.6.*

ESTB(q, B) is the estimated number of partial bindings of evaluating q given that variables in B are bound. These estimates, represented by functions ε , are computed based on a cheap preprocessing of the ontology making use of two structures:

Precompletion is used to get estimates for query atoms Ty, PV, SA, DF, like number of *named concepts for given individual*, or *number of same individuals*, as shown in Section 2.6. Note that precompletion is computed during the initial consistency check by the tableau algorithm that has to be performed before any query answering algorithm is executed.

Told Axioms are used to get estimates for all other SPARQL-DL^{NOT} atoms, except NOT and CORE. This includes *told concept hierarchy* (SCO, EC atoms) and *told role hierarchy* (SPO, EP atoms).

These estimates have to be done for all query atom types, as shown in Table B.1 in Appendix B.

Example 24 *As an example, $\text{ESTC}(\text{SCO} (?v, \text{Person}), \emptyset) = |CN|t_{CC}$ is equal to the estimated time for retrieving all (direct and indirect) named subconcepts of Person, where t_{CC} is an estimated time for computing a single consistency check of \mathcal{K} , as discussed in Section 2.6.*

Next, $\text{ESTB}(\text{SCO} (?v, \text{Person}), \emptyset)$ is the estimated number of subconcepts of Person, computed as the number of axioms of the form $A \sqsubseteq \text{Person}$.

The estimation strategy for SPARQL-DL^{NOT} query atoms, except NOT and CORE_γ, are computed based on told axioms, precompletion and estimated cost of basic reasoner API operations. Estimates for ESTC and ESTB of CORE_γ atoms, described in Table B.1, follow the course of the *simple evaluation strategy* used in line 8 of Algorithm 7.

Comparing to CORE_γ atoms, estimation of NOT(Q') atom evaluation costs is rather difficult, as their evaluation depends strongly on the shape of Q' . Thus, at present, this query atom type is excluded from query reordering optimizations and are evaluated always as last atoms in the query. Whenever more NOT(Q') atoms are present in a query, their mutual ordering is arbitrary.

Although the estimates introduced presented in full in Table B.1 in Appendix B are rough and neglect the particular query atom arguments, they are cheap to compute and – when used in static reordering (or dynamic reordering introduced in the following paragraph) – significantly improve typical query performance, as shown in Section 5.2.1. Still, the estimates do not take into account whether the ontology is classified (i.e. concept hierarchy computed), or realized (i.e. membership of individuals to named concepts computed), which are the operations that are often performed when handling large data sets. More in-depth statistical analysis would need to study the particular implementation/internal optimizations of tableau reasoner operations and will be part of my future work.

Dynamic Query Reordering. Static reordering introduced in the previous paragraph performs well whenever the number of query atoms is small (less than 8). As the size of the query grows, computing all permutations turns out not to be scalable. Furthermore, as the reordering method uses only rough estimates of knowledge base statistics, it is prone to cumulating statistical errors. These problems can be overcome by reordering query atoms dynamically, as shown in Algorithm 13.

Algorithm 13 Dynamic Query Reordering Function.

```

1: function DYNAMICNEXT( $\mathcal{K}, Q, \mu$ )
2:    $[q^*|Q_r] \leftarrow Q$  and  $cost^* \leftarrow \infty$ 
3:   for  $q \in Q_r$  do
4:     if  $cost^* > ESTC(\mu(q), \emptyset) + \alpha \cdot ESTB(\mu(q), \emptyset)$  then
5:        $cost^* \leftarrow ESTC(\mu(q), \emptyset) + \alpha \cdot ESTB(\mu(q), \emptyset)$ 
6:        $q^* \leftarrow \mu(q)$ 
7:   return  $[q^*|Q_r \setminus \{q^*\}]$ 

```

This method does not compute the best ordering of the query atoms in advance (and thus does not require exploring exponential number of all permutations of query atoms), instead it picks the best atom to be evaluated in the next execution step based on the actual variable binding, thus being similar to a state space search. The cost is evaluated as $ESTC(\mu(q), \emptyset) + \alpha \cdot ESTB(\mu(q), \emptyset)$, where α is a parameter (weight). While $\alpha = 0$ corresponds to a greedy search strategy, values of $\alpha > 0$ represent the weights with which the estimated cost of the rest of the query contributes to the atom cost.

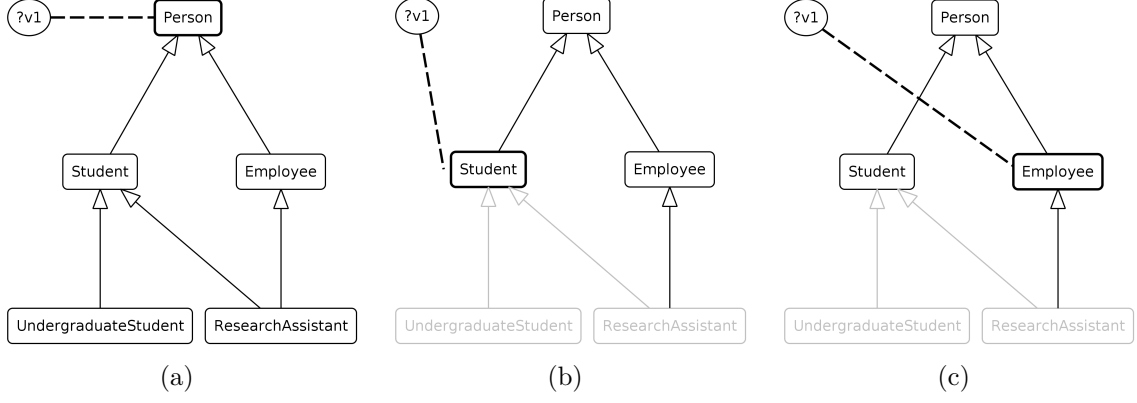


Figure 4.3.: Exploiting concept hierarchy for down-monotonic variable optimization in LUBM dataset.

Down-Monotonic Variables. Told concept and property hierarchies, discussed in previous paragraphs, can be reused also for pruning variable binding candidates as follows. Consider the query

$$[\text{SCO} (?v_1, \text{Person}), \text{Ty} (?v_2, ?v_1)]$$

which retrieves all individuals ($?v_2$) of the concept **Person** together with their actual type ($?v_1$). When evaluating this query in this order, it is possible to exploit information from the concept hierarchy and prevent variable $?v_1$ to be bound to named concepts that are subconcepts of concept **Person** which produced no result when used as a binding for $?v_1$.

Thus, on line 22 of Algorithm 10, named concepts taken from $\text{SUBC}(\mathcal{K}, \text{Person}) = \{\text{Person}, \text{Student}, \text{Employee}, \text{UndergraduateStudent}, \text{ResearchAssistant}\}$ are ordered according to the topological ordering with respect to the told concept hierarchy – first, the most general concepts are tested, and then their subconcepts, as demonstrated in Figure 4.3. Left-most subfigure shows the initial partial binding $\{?v_1 \mapsto \text{Person}\}$ that is contained in at least one solution. This makes possible that $?v_1$ binds to some of its subconcepts. Next, partial binding $\{?v_1 \mapsto \text{Student}\}$ is found. Since it is not contained in at least one solution, neither partial binding $\{?v_1 \mapsto \text{UndergraduateStudent}\}$ nor $\{?v_1 \mapsto \text{ResearchAssistant}\}$ need to be tested. Last, as **Employee** has no subconcepts that are not pruned, down-monotonic variable optimization cannot be used, no matter whether partial binding $\{?v_1 \mapsto \text{Employee}\}$ is contained in any solution or not.

Generalizing this example, at a given execution point *down-monotonic variable* is any variable $?v$ that occurs in an atom of the form $\text{Ty}(\bullet, ?v)$, or $\text{PV}(\bullet, ?v, \bullet)$ later in the query Q . Whenever the engine is about evaluating an atom q that contains a down-monotonic variable $?v$, it can safely perform the above described pruning.

5. Software Implementation

As a part of this thesis, I developed several software tools that show feasibility of the methodology and framework proposed in Chapter 4. In this Chapter I will discuss generic software tools that implement ideas and techniques introduced in the previous Chapter, while in Chapter 6 I will discuss applications that use these generic tools. All tools discussed in this chapter are released under an open-source license and involve

Java OWL Persistence API (JOPA) that implements the programmatic access to OWL ontologies and implements techniques and ideas from Section 4.1.

Pellet SPARQL-DL engine that is an implementation of SPARQL-DL, a significant subset of the SPARQL-DL^{NOT} language described in Section 4.2. This engine has been part of the Pellet distribution since 2008.

OWL2Query that is an implementation of SPARQL-DL^{NOT} using techniques presented in Section 4.2. This engine is generic and can be used with any OWL 2 DL reasoner (e.g. through OWLAPI [57]).

5.1. Java OWL Persistence API

The reference implementation of the Ontology Persistence Layer is the Java Ontology Persistence API (JOPA) ¹ following the architecture presented in Figure 5.1. Due to different semantics of regular axioms and integrity constraints (see Section 2.8), the integrity constraints must be kept separately from the ontology to prevent OWL reasoners from handling them as OWL axioms with standard OWL 2 DL semantics. Integrity constraints for different applications can be kept in a single OWL document and distinguished by OWL annotations with `isIntegrityConstraintFor` annotation property, filler of which is the application identifier of the respective application, i.e.

$$constraint \text{ @ isIntegrityConstraintFor 'A'}$$

says that *constraint* is interpreted as an integrity constraint for the application **A**. For example, OWL integrity constraint 5.1 from the following example would be assigned to application **StruFail**, ver. 0.1 as follows:

$$(\text{Failure} \sqsubseteq \forall \text{isFailureOf} \cdot \text{Structure}) \text{ @ isIntegrityConstraintFor 'StruFail - 0.1'}$$

or in RDF/XML syntax in OWL as depicted in Figure 5.2

¹see <http://krizik.felk.cvut.cz/km/jopa>, cit. 12/10/2011

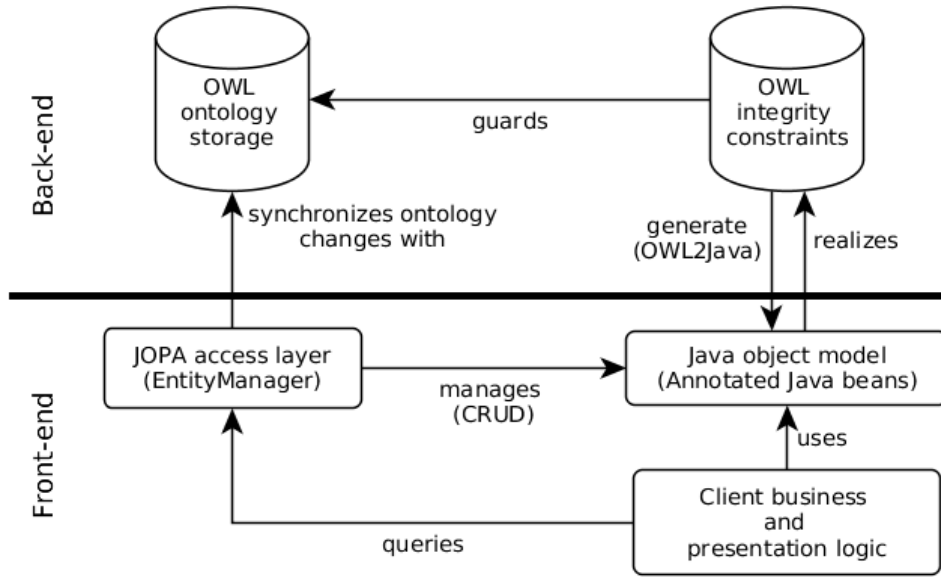


Figure 5.1.: JOPA overall architecture.

From the application developer point of view, the API of JOPA is similar to JPA 2.0 (see [106] for the description of JPA 2.0). In the object model, Java beans (called 'entities' in the rest of this thesis) represent OWL classes, while their instances (called 'entity instances') correspond to OWL named individuals. Java annotations of entities and their properties express the ontology – object model contract introduced in Section 4.1.1. With respect to the open world assumption nature of OWL 2 DL semantics, it is important to make a distinction between Java properties:

- value of which is inferred² by an OWL 2 reasoner. To avoid value overriding (and thus possibly also ontology inconsistency) their modification by client code is prohibited,
- value of which is not inferred and thus can be freely modified by the application. Values of these properties are loaded/stored directly from/to OWL axioms.

The persistence of entity instances is achieved by a transactional entity manager, where transactions are handled as described in Section 4.1.2. Each entity instance has very similar life-cycle as JPA 2.0 entity instances. The entity manager provides several operations for managing life-cycle of instances (new, managed, detached, removed)³ of the annotated Java beans similarly to JPA 2.0 entity managers, above all:

- *persist* operation for a new (resp. *merge* operation for a detached) entity instance persists it by propagating corresponding changes in terms of newly added/removed OWL axioms into the ontology,

²See `inferred=true` attribute in Figure 5.3.

³Readers not familiar with JPA instance states will find details in [106].

```

<?xml version="1.0"?>
<!DOCTYPE rdf:RDF [
<!ENTITY rdfs "http://www.w3.org/2000/01/rdf-schema#">
<!ENTITY ic "http://krizik.felk.cvut.cz/ontologies/2011/ic-example.owl#">
]>
...
  <owl:Class rdf:about="&ic;Failure">
    <rdfs:subClassOf>
      <owl:Restriction>
        <owl:onProperty rdf:resource="&ic;isFailureOf"/>
        <owl:allValuesFrom rdf:resource="&ic;Structure"/>
      </owl:Restriction>
    </rdfs:subClassOf>
  </owl:Class>
  <owl:Axiom>
    <isIntegrityConstraintFor>StruFail-0.1</isIntegrityConstraintFor>
    <owl:annotatedSource rdf:resource="&ic;Failure"/>
    <owl:annotatedProperty rdf:resource="&rdfs;subClassOf"/>
    <owl:annotatedTarget>
      <owl:Restriction>
        <owl:onProperty rdf:resource="&ic;isFailureOf"/>
        <owl:allValuesFrom rdf:resource="&ic;Structure"/>
      </owl:Restriction>
    </owl:annotatedTarget>
  </owl:Axiom>
...

```

Figure 5.2.: Integrity Constraint Serialization in OWL using RDF/XML syntax.

- *refresh* operation reloads the state of an entity instance from the ontology, and
- *remove* operation removes immediately the entity instance from the object model and all related axioms from the ontology at the transaction commit.

5.1.1. Object-Ontology Mapping in Java

Entities are automatically generated directly from the respective set of integrity constraints using the OWL2Java tool which is a part of the JOPA distribution. OWL2Java transforms the integrity constraints to Java objects according to the rules compliant with meta-rules 1-3 introduced in the paragraph 'compile-time' constraints in Section 4.1.1.

Example 25 *Let's consider the following integrity constraints for the StruFail application, ver. 0.1, described in Section 6.1 (the respective `isIntegrityConstraintFor` annotations are omitted for the sake of brevity):*

$$\text{Failure} \sqsubseteq \forall \text{isFailureOf} \cdot \text{Structure}, \quad (5.1)$$

$$\text{Failure} \sqsubseteq \leq 1 \text{ isFailureOf}, \quad (5.2)$$

$$\text{Failure} \sqsubseteq \geq 1 \text{ isFailureOf}, \quad (5.3)$$

$$\text{Failure} \sqsubseteq \forall \text{hasFactor} \cdot \text{Factor}. \quad (5.4)$$

*The integrity constraint (5.1) specifies that whenever a **Failure** instance is a subject of the relation `isFailureOf`, the object of the relation must be an instance of **Structure**. Next, each **Failure** instance must be in relation `isFailureOf` at least once (5.3) and at most once (5.2). Last, each **Failure** instance might be connected through the relation `hasFactor` only to instances of **Factor** (5.4). These integrity constraints are elaborated next in this section.*

As an example, Figure 5.3 shows an entity corresponding to the integrity constraints (5.1)-(5.4). The Java annotation in line 1 says that the Java class **Failure** is a representation of the OWL 2 DL class **Failure** (the `@OWLClass` annotation defines the full Internationalized Resource Identifier (IRI) of the respective OWL 2 DL class). Lines 3-10 are independent of the particular integrity constraints and thus do not influence the object-ontology contract. Lines 3 and 4 make a value⁴ of `rdfs : label` of each **Failure** instance accessible to the application logic. Lines 5 and 6 make read access to IRIs of all inferred types for the given entity instance (e.g. each failure record is represented as an OWL individual being an instance of the **Failure** class. But this individual might also belong to the OWL class **Event** that is not a part of our example set of integrity constraints.). Similarly, lines 9 and 10 provide read access to inferred values of all OWL object and data property relationships, subject of which is the OWL individual corresponding to the actual **Failure** instance. Lines 7 and 8 define an identifier of the entity instance. The distinction between inferred and not inferred properties makes it

⁴ In OWL 2, each string literal value can be expressed in different language/regional variants. The actual language is selected during instantiating the entity manager. Thus, each annotation property value and data property value is loaded and stored in the actual language.

```

1  @OWLClass(iri = V.Failure)
2  public class Failure {
3      @OWLAnnotationProperty(iri = CV.RDFS_LABEL)
4      protected String label;
5      @Types(inferred=true)
6      protected Set<String> types;
7      @Id(generated = true)
8      protected String id;
9      @Properties(inferred=true)
10     protected Map<String,Set<String>> properties;
11     @OWLObjectProperty(iri = V.hasFactor)
12     protected Set<Factor> hasFactor;
13     @OWLObjectProperty(iri = V.isFailureOf)
14     @ParticipationConstraints({
15         @ParticipationConstraint(
16             owlObjectIRI = V.Structure,
17             min = 1,
18             max = 1)
19     })
20     protected Structure isFailureOf;
21     // setters/getters and other constraints omitted
22 }

```

Figure 5.3.: An example of generated Java model entity. **V**, **CV** are generated classes that serve as vocabularies.

possible to use an OWL individual IRI (**generated=true** causes the entity manager to generate a fresh identifier for an entity instance that has no correspondence to an OWL individual yet) as the identifier of the corresponding entity instance. This decision cause that multiple entity instances can share the same IRI (i.e. a **Failure** instance and an **Event** instance can refer to the same OWL individual). In this case, the entity manager is responsible for ensuring that at most one entity instance from the set of entity instances sharing the same IRI is persisted. If not, the entity manager rolls back the current transaction.

Lines 11-20 represent the mapping between the above mentioned integrity constraints and the Java object model. Lines 13 and 20 represent directly the integrity constraint (5.1). The Java annotation in line 13 defines the correspondence between the OWL 2 DL property **isFailureOf** and the Java property **isFailureOf**. Line 20 defines the type of the Java property **isFailureOf** as **Structure**, which is a Java representation of an OWL 2 DL class **Structure**. Due to the strong typed character of the Java programming language, (5.1) is ensured *compile-time*, see Section 4.1.1.

The integrity constraints (5.2) and (5.3) are expressed by the Java annotation in lines 14-19. This annotation defines the correspondence between the Java class **Structure** and the OWL 2 DL class **Structure** specifying its full IRI (see line 16), the minimal cardinality (see `min=1` in line 17) and the maximal cardinality (see `max=1` in line 18) of the **isFailureOf** relationship. As the Java property **isFailureOf** is not a collection-valued property but a single-valued property of type **Structure**, the specification of the maximal cardinality is superficial here and could be omitted. However, the specification of the minimal cardinality is important here and represents an integrity constraint with *run-time* evaluation strategy, see section 4.1.1.

The integrity constraint (5.4) is reflected by the Java annotation in line 11, which establishes the correspondence between the **hasIssue** Java property and the **hasFactor** OWL 2 DL property. As there is no constraint on minimal cardinality of **hasFactor**, the Java property **hasFactor** is a collection-valued property (a **Set** in particular). The generic type **Set<Factor>** is used to ensure (5.4) in compile-time.

5.1.2. Transactional Processing in Java

The reference Java implementation JOPA implements the overall operation cycle depicted in Figure 4.1 in the following way. Each transaction request causes a new Java thread to be created, which serves the whole operation cycle. The activity FR5 is a general activity that can be executed either by the *persist/merge* operation of the entity manager or by changing a property of a managed entity bean. In the latter case, AspectJ run-time code instrumentation [115] is used to schedule the ontology changes and detect preliminary compile-time and run-time integrity constraint violations, as well as lazy fetching of properties. Validation of compiled integrity constraints (see FA2) is ensured automatically by the Java run-time, whereas the runtime integrity constraints (see FA3) need to be verified by means of validators automatically generated from the respective integrity constraints.

If the user wishes to save the changes (flow starting at FR4), the transaction gets to close. Even if the compile-time and run-time constraints have been checked at the moment, it is necessary to verify the consistency of the modified ontology (see BA3). If the ontology is consistent, it is necessary to verify integrity constraints of all applications using the ontology. This is done by asking DCQ^{NOT} queries representing the respective integrity constraints (see Section 2.8). Advanced aspects of transactional support of the JOPA entity manager have been studied and implemented by Martin Ledvinka in his bachelor's thesis [116] under my supervision – his prototype is currently being tested and I plan to integrate it into the standard JOPA distribution.

There are currently two back-end OWLAPI-based [57] implementations: (i) a simple implementation that accesses ontologies in OWL files, and (ii) a database-backed implementation that uses OWLDB [117] to store OWL ontologies in a relational database. In both variants, any OWLAPI-compliant OWL 2 DL reasoner can be used to check consistency, validate integrity constraints and evaluate SPARQL-DL^{NOT} queries (flow from FR2 to FS3 in Figure 4.1), although Pellet [32] is currently preferred as it provides built-in optimized DCQ^{NOT} support. In case that another OWL 2 DL reasoner is used,

SPARQL-DL^{NOT} engine OWL2Query, described in Section 5.2, can be used on top of it to validate integrity constraints.

5.2. SPARQL-DL engine in Pellet and OWL2Query

Originally, I have implemented the evaluation and optimization techniques presented in Section 4.2.2 for SPARQL-DL and my implementation is currently integral part of Pellet [55], one of the most widely used OWL 2 DL reasoners.

To allow other OWL 2 DL reasoners to make use of the expressive querying functionality, I generalized and extended the original Pellet engine towards SPARQL-DL^{NOT}. This open-source implementation is called OWL2Query⁵ and can be used

- as a standalone query engine
- as a part of the JOPA distribution
- as a Protégé 4.1 plug-in that contains, in addition to OWL2Query, also SPARQL-DL^{NOT} query visualization component developed by Bogdan Kostov in his Master's thesis [118] under my supervision.

Both Pellet and OWL2Query engines accept SPARQL-DL^{NOT} queries in SPARQL syntax⁶. Additionally, queries can be constructed programmatically in the similar manner as Java Persistence Query Language (JPQL) queries in JPA.

5.2.1. Evaluation of the Query Engine

Experiments in this section will show overall execution time of selected benchmark queries to

- compare my SPARQL-DL implementation to another recent OWLAPI-based SPARQL-DL implementation presented in [119] (denoted *Derivo* in the next paragraphs),
- demonstrate efficiency of the evaluation technique presented in Section 4.2.2 using Pellet query engine (SPARQL-DL queries), OWL2Query engine backed with Pellet reasoner and OWL2Query engine backed with JFact reasoner,
- present benefits of the core evaluation technique, static, dynamic query reordering and down-monotonic variable optimization described in Section 4.2.2.

As none of the use cases presented in Chapter 6 provides dataset that is large enough for demonstrating the efficiency of the evaluation technique, two benchmark datasets will be used in this section:

⁵see <http://krizik.felk.cvut.cz/km/owl2query>, cit. 12/10/2011

⁶negation as failure support is added by SPARQL 1.1. that is currently being standardized

- LUBM ontology [69] has expressiveness $\mathcal{SHI}(D)$ (subset of OWL DL). The ontology has a dominant ABox part with approx. 17000 individuals in LUBM(1) while the TBox consists of just several tens of classes and properties.
- UOB ontology [120] is a $\mathcal{SHOIN}(\mathcal{D})$ extension of LUBM.

Both of the benchmarks are synthetic and describe the domain of universities, students, teachers, courses and their mutual relationships. The UOB vocabulary slightly differs from the LUBM vocabulary. First, both ontologies have different namespaces that are omitted in the following paragraphs to improve readability. Second, some classes and properties have different names, in our case **memberOf** and **advisor** in LUBM correspond to **isMemberOf** and **isAdvisedBy** in UOB.

The LUBM dataset comes with 16 pure conjunctive ABox queries without undistinguished variables with different characteristics (low vs. high selectivity, small vs. large input, etc.). In a similar fashion I constructed 10 SPARQL-DL^{NOT} queries, presented in the next section, some of which are extensions of the original conjunctive ABox queries, making use of their benchmarking properties, like selectivity. Besides these novel SPARQL-DL^{NOT} queries, for testing dynamic query reordering and core evaluation strategy, further specialized queries were created.

Before executing each of the queries in the subsequent sections an initial consistency check of the ontology is performed to warm-up the respective OWL 2 DL reasoner and to create the completion and pre-completion data structures as described in Section 2.6.

The consistency check execution time ranges from 800 ms to 1100 ms for LUBM(1) and from 1800 ms to 2100 ms for UOB(1). All tests were run on Intel(R) Core(TM)2 CPU 6400 at 2.13GHz with 3GB RAM. All results are averages over 10 independent runs of the query execution.

5.2.2. Performance of Different Engine Implementations

For benchmarking the overall performance of SPARQL-DL^{NOT} implementations, the following queries over LUBM/UOB datasets were created. These queries are modified versions of original LUBM benchmark queries :

EQ_1 is the query presented in Example 8,

EQ_2 is the query presented in Example 9,

EQ_3 is the query presented in Example 10,

EQ_4 (mixed ABox + TBox + RBox query)

“Which courses ($?v_2$) and of which type ($?v_3$) is the GraduateStudent5 related to and what kind of relation ($?v_1$) it is.”

$[PV(GraduateStudent5, ?v_1, ?v_2), Ty(?v_2, ?v_3), SCO(?v_3, Course)],$

EQ₅ (Query with transitive property atom))

“What are the fillers of a transitive property for Department0 ”

[PV (Department0, ?v₁, ?v₂), Trans (?v₂)],

EQ₆ (Modified version of *EQ₄* - DisjointWith)

“Which graduate students (?v₁) are related to some course (?v₃) that is not a GraduateCourse. What kind of relationship (?v₂) and course (?v₄) it is ?”

[Ty (?v₁, GraduateStudent), PV (?v₁, ?v₂, ?v₃), Ty (?v₃, ?v₄), DW (?v₄, Course)],

EQ₇ (mixed ABox + TBox query)

“Which persons (?v₂) and of which type (?v₁) teach a course (?v₃) that they take at the same time ?”

[SCO (?v₁, ⊤), Ty (?v₂, ?v₁), PV (?v₂, takesCourse, ?v₃),
PV (?v₂, teachingAssistantOf, ?v₃)],

EQ₈ (mixed ABox + TBox query)

“Which persons (?v₁) and of what type (?v₃) has an advisor (?v₂)”

[PV (?v₁, advisor, ?v₂), Ty (?v₁, ?v₃), SCO (?v₃, Person)]

EQ₉ (mixed ABox + TBox query)

“Which persons (?v₁) of which type (?v₂) are teaching assistants of a course (?v₃)”

[SCO (?v₂, Person), Ty (?v₁, ?v₂), PV (?v₁, teachingAssistantOf, ?v₂), Ty (?v₂, Course)]

EQ₁₀ (Query with negation)

“Which graduate students (?v₁) are related (?v₂) to any Employee (?v₃) who is not a full professor”

[Ty (?v₁, GraduateStudent), PV (?v₁, ?v₂, ?v₃), Ty (?v₃, Employee),
NOT ([Ty (?v₃, FullProfessor)])]

Results of evaluating these queries are depicted in Table 5.1. The results show that my original implementation of the Pellet SPARQL-DL query engine is the most efficient one – this is not surprising, as the query engine works directly in the Pellet internal model and does not need to perform OWLAPI transformations like in the case of both OWL2Query and Derivo implementations.

Another interesting result is that OWL2Query significantly outperforms Derivo on all benchmark queries, due to the reordering optimizations sketched in Section 4.2.2. Another important conclusion is that Pellet remains more efficient than JFact, due to internal caching of completion trees and other optimizations.

The times for evaluation the query using the down-monotonic variables optimization (see Section 4.2.2) are comparable except the query *EQ₇*, in which case the query evaluation time reduces significantly (e.g. in case of the pure Pellet implementation from approx 300ms to approx. 70ms) due to the pruning of the down-monotonic variable ?v₁.

query	results	P	P_{DM}	$O2Q + P$	$O2Q + J$	$Der + P$	$Der + J$
EQ_1	4145	1430	1397	7961	> 15min.	23380	> 15min
EQ_2	1094	53	44	3549	821919	10672	> 15min
EQ_3	761	46	70	1497	> 15min.	5463	> 15min
EQ_4	3	182	171	1929	> 15min.	3912	> 15min
EQ_5	1	34	55	682	> 15min.	6040	> 15min
EQ_6	0	35	33	246	200	255	3567
EQ_7	0	303	70	7781	2183	11219	> 15min
EQ_8	10804	213	199	3678	2162	9960	> 15min
EQ_9	1628	104	123	3133	752	6688	> 15min
EQ_{10}	11031	N/A	N/A	23789	30007	N/A	N/A

Table 5.1.: Performance of query evaluation over LUBM(1). $O2Q$ resp. Der mean $OWL2Query$, resp. $Derivo$ query engines, and P , resp. J means Pellet, resp. JFact tableau reasoner and P_{DM} means Pellet with down-monotonic variable optimization. Next, $results$ denotes number of results of the query. All times are in milliseconds.

5.2.3. Performance of the Dynamic Reordering Method

Neither the queries introduced in the previous section nor the benchmark queries of UOB are long enough to be able to show benefits of the dynamic reordering method. Thus, I had to create a fresh query set for these experiments.

Example 26 (A long query) *The following query DQ_7 retrieves all graduate students that are taught by, have a common publication with and are teaching with someone from the same university part. It also retrieves the respective courses, publications and university parts.*

[Ty (University, ?vw), PV (subOrganizationOf, ?vr, ?vw),
PV (memberOf, ?v₁, ?vr), PV (memberOf, ?va, ?vr),
Ty (GraduateStudent, ?v₁), PV (takesCourse, ?v₁, ?v₃),
PV (teacherOf, ?va, ?v₃), PV (teacherOf, ?va, ?vz₂),
PV (advisor, ?v₁, ?vb), PV (teachingAssistantOf, ?v₃, ?vz₂),
PV (publicationAuthor, ?vq, ?v₁),
PV (publicationAuthor, ?vq, ?va)].

If the actual binding for the publication is not important, a slightly modified version of this query introducing an undistinguished variable would be DQ_8 , where ?vq is replaced with !uq.

Performance of the dynamic reordering strategy for the UOB(1) benchmark queries (presented in [120]) and DQ_7, DQ_8 is shown in the Table 5.2, but only for queries that

Query	results	atoms*	static [ms]	dynamic [ms]	no [ms]
UQ_1	32	2	220	150	470
UQ_3	666	2	210	230	380
UQ_4	383	3	560	650	390
UQ_5	200	2	320	230	550
UQ_8	303	2	5620	3430	6040
UQ_9	1057	4	2840	530	570
UQ_{11}	1930	5	1100	530	> 10min.
UQ_{12}	65	3	210	380	530
UQ_{13}	379	2	> 10min.	> 10min.	> 10min.
UQ_{14}	6893	4	250780	232720	> 10min.
DQ_7	2	12	> 10 min.	8910	> 10 min.
DQ_8	2	11	> 10 min.	3540	> 10 min.

Table 5.2.: Performance of the dynamic reordering using the Pellet SPARQL-DL engine over the UOB dataset. Each UQ_x denotes a corresponding query from the UOB DL benchmark. *atoms** denotes number of query atoms after performing the domain/range simplification, *results* denotes number of results of the query and *static*, *dynamic* and *no* denote query execution times for these reordering strategies.

have (after performing the domain/range simplification presented in [71]) at least 2 atoms so that a reordering method becomes applicable. In all cases, 10% of individuals were taken for computing the statistics and estimating query atom costs. We can see that static and dynamic reordering (for $\alpha = 0$) have similar performance for short queries, while the dynamic reordering significantly overtakes the static one for the longer ones.

The significance of using a query reordering method is shown in the column *no* that presents query execution times when the query is not reordered (i.e. for the default ordering). For some queries the query execution times are more than an order of magnitude worse than their static/dynamic reordering counterparts, while for others are comparable to the reordering strategies.

5.2.4. Performance of the Undistinguished Variables Optimizations

Since neither of the benchmarks contains conjunctive queries with undistinguished variables a new query set was created. The query set does not contain any query that is itself just a single core, since for those queries the core evaluation performance is the same for both the original execution and the core evaluation strategy. As can be observed from the nature of the core evaluation strategy presented in Section 4.2.1, Algorithm 6 is beneficial only for queries that contain some non-core atoms.

Consider the following set of queries:

Q_1 is the query presented in Example 11,

LUBM(1)				UOB(1)			
	results	NB	time [ms]	results	NB	time [ms]	engine
Q_1	208	622673 30028	74030 1800	184	569650 54562	132020 5100	<i>simple</i> <i>core</i>
Q_2	1621	875340 3249	5180 910	2256	1573345 4517	10510 2280	<i>simple</i> <i>core</i>
Q_3	1099	924238 24216	26130 1320	2262	1581081 46838	15010 3760	<i>simple</i> <i>core</i>
Q_4	0	541 541	330 310	0	878 878	800 380	<i>simple</i> <i>core</i>
Q_5	208	4998812 30028	38150 2360	184	6015688 54562	45512 4720	<i>simple</i> <i>core</i>

Table 5.3.: Performance evaluation of the core strategy of the Pellet SPARQL-DL reasoner over the LUBM(1) and UOB(1) DL dataset. The rows labeled *simple* denote the simple evaluation strategy as described in Example 11, while the rows labeled with *core* denote the evaluation using cores. The query evaluation took *Time* ms (without the initial consistency check), *results* denotes the number of bindings valid for the query and $NB = (N_{IC} + |IN|N_{IR})/10^3$, where N_{IC} is the count of *IC* calls and N_{IR} is the count of *IR* calls.

Q_2 (An acyclic query with one undistinguished variable)

“Retrieve all teachers of some course that is taken by at least one student.”

$[PV(?v_1, \text{teacherOf}, ?v_2), PV(!u_3, \text{teacherOf}, ?v_2)],$

Q_3 is the query presented in Example 7,

Q_4 is the same as Q_3 , but with $?v_2$ replaced by $!u_2$,

Q_5 (A query with two undistinguished variables, one of which is in a cycle – a modified version of Q_1 with $?v_3$ replaced with $!u_3$.)

“Get all students that are members of some part of some organization and whose advisor teaches a course they take. Get also the courses.”:

$[PV(?v_1, \text{advisor}, !u_4), PV(!u_4, \text{teacherOf}, ?v_2), Ty(!u_4, \text{Employee}),$
 $PV(?v_1, \text{takesCourse}, ?v_2), PV(?v_1, \text{memberOf}, !u_3)],$

The test results are shown in Table 5.3. Both the LUBM and UOB performance results show that the more distinguished variables the query contains, the worse the performance of the original execution is and that the core optimization technique overtakes the optimized strategy by at least an order of magnitude.

The importance of undistinguished variables is presented by queries Q_3 and Q_4 . For both *LUBM* and *UOB* the query Q_4 has no results, since there is no person working for

an explicitly asserted **ResearchGroup**. On the other hand, Q_3 that matches also inferred **ResearchGroup** instances has more than 1000 results in both cases.

6. Use Cases

The proposed methodology and its implementation JOPA is currently used in two applications. One of them is StruFail, a knowledge base of structural failures, that serves as a bank of structural damage cases due to floods. Semantic web ontologies are used in StruFail to allow posing complex queries, as well as specifying user's knowledge to various levels of details and granularity. In the next section, I will discuss the use of the methodology together with expressive queries in StruFail, that was designed as a part of this thesis. StruFail business and UI logic was implemented as a part of the master's thesis of Jan Abrahamčík [121] under my supervision.

The other system is SHM-CMS [122], an intelligent ontology-based content management system for the domain of risk assessment in construction engineering, describing content, its types, dissemination and their relation to the constructions. The system is currently under development in cooperation with VCE, Austria and Johannes Kepler Universität Linz, Austria within the EU-funded project IRIS ¹. I will not describe this application in detail as I have not been involved in its design and development. Instead I refer the reader to the bachelor's thesis of Jiří Kopecký [123], who implemented an initial version of this system based on the methodology introduced in this thesis.

6.1. StruFail System

The main objective of the StruFail system is to provide a common platform for collecting and sharing knowledge of previous documented failures, and sharing understanding what went wrong and which protective measures worked best, among professionals from various field of expertise. This allows professionals in the field to consciously rely on a sound experience based on previous cases examined by others.

The system was developed in cooperation with the Institute of Theoretical and Applied Mechanics of the Czech Academy of Sciences within the EU-funded project CHEF ² and is freely available online³. Details on system usage and scope can be found in [34]. StruFail is built on top of the Java EE platform, deployed on the Glassfish application server ver. 2 [124]. All libraries used by the system are licensed under some type of open-source license.

¹see <http://www.vce.at/iris>, cit. 5/1/2011

²see <http://www.chef.bam.de>, cit. 5/1/2011

³see <http://www.itam.cas.cz/strufail>, cit. 15/1/2011

6.1.1. System Design

During the initial phase of system design, the ontology of structural failures⁴ was iteratively defined (for the purpose of comparing subsequent ontology versions, I designed a tool OWLDiff that is briefly described in Appendix C) with cooperation of the community of civil engineering experts. The ontology describes relationships between structures, failures and their manifestations together with respective taxonomies. Let's recall the fundamental parts of the ontology that are necessary for understanding the rest of this thesis.

The civil engineering experts agreed on the fundamental notions that resulted in the skeleton of the domain ontology, including classes **Structure**, **Component**, **Failure**, **Manifestation** and **RawMaterial** and relationships between them, part of which is schematically depicted in Figure 6.1.

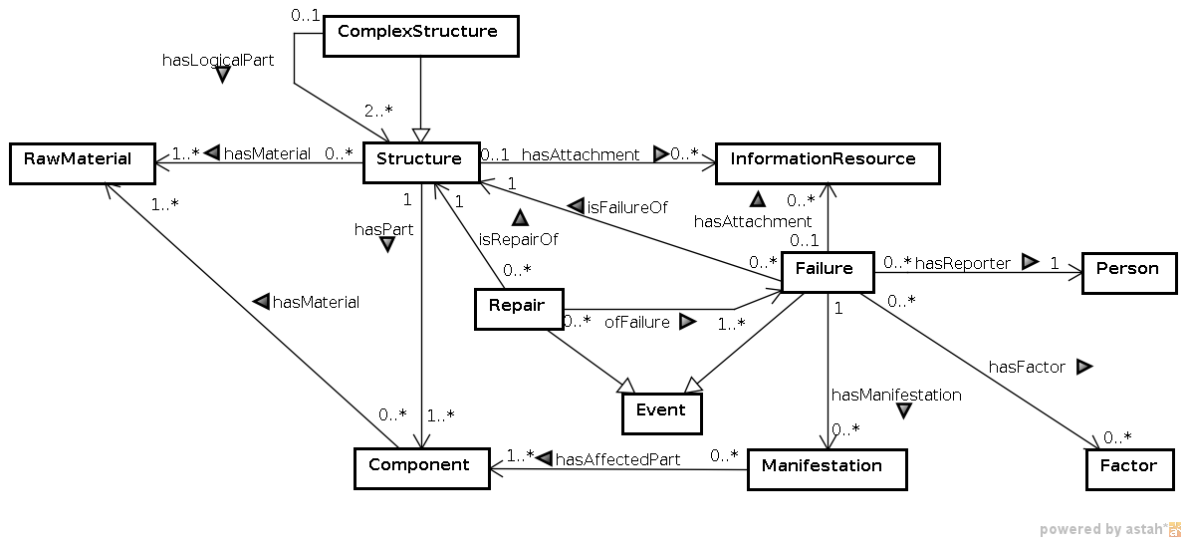


Figure 6.1.: Fundamental Parts of the StruFail Ontology.

A vertex-edge-vertex triple represents a (local) domain restriction and (local) range restriction for an OWL object property. E.g., the edge *isFailureOf* going from *Failure* to *Structure* depicts an axiom of the form

$$\text{Failure} \sqsubseteq \forall \text{isFailureOf} \cdot \text{Structure}.$$

UML relational multiplicity of each edge denotes number restrictions. E.g. axiom

$$\text{Failure} \sqsubseteq (= 1 \text{ isFailureOf})$$

represents the multiplicity 1 at the end of the *isFailureOf* edge.

⁴see <http://krizik.felk.cvut.cz/ontologies/2009/failures.owl>, cit.12/10/2011.

The dominant notion in the OWL ontology is the **Structure** class. Each instance of this class represents a particular construction (e.g. “Charles Bridge in Prague”). Each structure can have assigned one or more failure records (e.g. “Salination of building material”). In fact, each **Failure** instance integrates one or more failure **Manifestations** that occur with respect to the same factors (e.g. “A flood can cause partial collapse of some walls of a structure, as well as moisture stains occurrence afterwards”). Each failure **Manifestation** is connected to one or more **Components** (e.g. “foundations, wall”) that were affected by the failure. A **Component** can bear information about **RawMaterial** it was built from. Many of the shown classes are natural roots of rich specialization hierarchies, namely structure types, components, materials, or factors. Due to their size (most hierarchies contain tens to hundreds of classes), let’s sketch just an example of them. Figure 6.2 shows a structure sub-hierarchy of different types of religious objects and their subtypes.

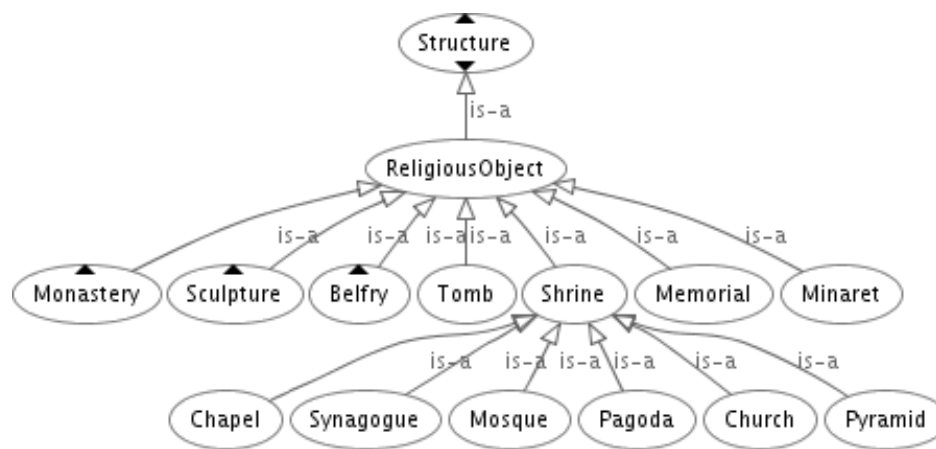


Figure 6.2.: Part of the Structure Taxonomy.

These hierarchies are beneficial for the StruFail users both (i) when reporting a failure – to specify their experience to different levels of accuracy (e.g. either **Pyramid** or **Shrine** can be used to specify a part that is affected by a **Failure**), and ii) when exploring the knowledge base – to specify queries to different levels of accuracy (e.g. one can query the knowledge base to get typical damages that occur on pyramids, or on shrines in general.)

On the other hand, definition of taxonomical knowledge as well as axiomatization of complex relationships, e.g. types of materials to be used for different components (parts of the structure), turned out to be much more problematic and time-consuming task for them.

Traditional Design

As the ontology content was not guarded by integrity constraints, the development of StruFail was slow, as dynamic changes of attribute values and relationships in the ontology by civil engineering experts obsoleted the developed user interface logic (see Figure 6.4

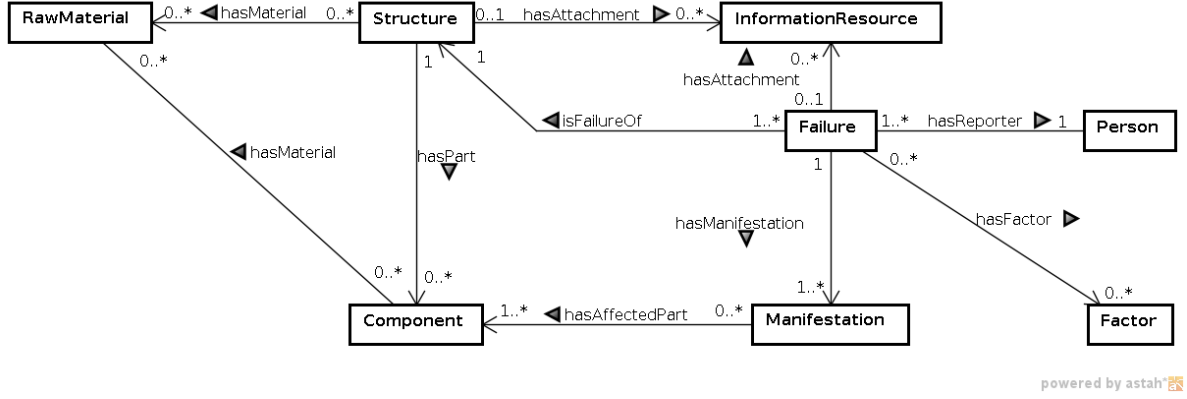


Figure 6.3.: Integrity constraint sets for StruFail application. Each vertex corresponds to an OWL 2 DL class and an edge corresponds to an OWL 2 DL object property.

and Figure 6.5) that crashed in runtime. E.g. some failure records were missing the `isFailureOf` link to the corresponding structure records. Furthermore, these user interface logic crashes didn't point us to the problematic failure records that caused the crash, which prolonged error debugging. This highlighted the need of contract introduced by JOPA.

Due to the missing transactional support, every time the user wanted to see details of a failure in the user interface, ontology consistency check had to be performed by the underlying OWL reasoner Pellet before retrieving the respective failure record from the ontology, because the failure record could already have been changed by another user. Even for the ontology of structural failures containing several tens of failure records, the consistency check took approx. 1 second, which significantly delayed the responses of the user interface.

Design using JOPA

Using JOPA, a set of integrity constraints for StruFail was defined, based on the relevant parts of the ontology, and user requirements. Main parts of the integrity constraint set are presented in the UML class diagram in Figure 6.3. The visualization of integrity constraints is analogous to the visualization of ontological axioms in Section 6.1.1. I.e. each integrity constraint of type β_1 in Section 2.8 is represented by a vertex-edge-vertex triple. For example (5.1), i.e. the integrity constraint $\text{Failure} \sqsubseteq \forall \text{isFailureOf} \cdot \text{Structure}$, is rendered as an edge labeled `isFailureOf` from the `Failure` class to the `Structure` class. Each integrity constraint of type β_2 , β_3 , or β_4 is represented by the respective UML relational multiplicity, e.g. integrity constraints (5.2) and (5.3) in Section 5.1, i.e. $\text{Failure} \sqsubseteq \geq 1 \text{ isFailureOf}$, and $\text{Failure} \sqsubseteq \leq 1 \text{ isFailureOf}$.

Relationship between the Ontology and the Integrity Constraints Let's look closer at the relationship between the schema in Figure 6.1 and the schema in Figure 6.3. They look similar on the first sight, but there are substantial differences between them. While Figure 6.1 depicts ontological relationships (i.e. interpreted under open world assumption) that are valid for the whole domain and are used for inferencing (and thus also in query answering), Figure 6.3 depicts only *minimal* set of integrity constraints necessary for the business logic of StruFail that serve for object model synchronization and thus data validation. The ontological knowledge depicts three⁵ OWL classes **ComplexStructure**, **Repair** and **Event** that are not present in the integrity constraint set – these classes are available only for inferencing and query answering in the UI and are not part of the StruFail object model.

Note that in many cases the relationships multiplicities differ in the ontological description and in the integrity constraint set. For example, in the ontological knowledge, each **Component** is related through **hasMaterial** property to at least one **RawMaterial**, it is made of (depicted in Figure 6.1 as multiplicity 1..* at the end of **hasMaterial** edge from **Component** to **RawMaterial**). This is valid (using open-world assumption), no matter whether the particular material is known or not. On the other hand, integrity constraint set does not contain such restriction, as the StruFail business logic considers as valid also **Component** instances without explicitly assigned **RawMaterial** instances (depicted in Figure 6.3 as multiplicity 0..*) – in this case the integrity constraint is less restrictive than the ontological description. Note, that **Component** records without assigned materials neither cause inconsistency of the ontology (because material assignment is inferred), nor invalidate the integrity constraints (because material assignment is not compulsory).

A different example is the following one. In the ontological knowledge, some people (instances of the class **Person**) might be reporters of a failure, while other need not (this situation is depicted in Figure 6.1 as multiplicity 0..* at the end of the **hasReporter** edge). On the other hand, a person is relevant to the StruFail application only if (s)he is a failure reporter (depicted in Figure 6.1 as multiplicity 1..*) – in this case the integrity constraint is more restrictive than the ontological description. Note, that in this case **Person** records without assigned failure reports do not cause ontology inconsistency, but cause integrity constraint validations for StruFail.

Integrity Constraint Benefits After the definition of integrity constraints, the Java object model was generated based on the integrity constraints, as described in Section 5.1 and UI logic (see Figure 6.4 and 6.5 in the next section) was developed on top of it. Simultaneously, the ontology was edited by civil engineering experts to refine taxonomies of structures, failure manifestations and materials, define complex relationships and insert new (or modify existing) failure records. Several times, this concurrence in the ontology access resulted in a clash of the data inserted by Protégé and the developed

⁵These are only examples to demonstrate the differences between ontological and integrity constraint descriptions. The developed StruFail ontology is more complex and contains many additional classes, object properties and their axiomatization, all of which is omitted here for the sake of readability, but is available at <http://krizik.felk.cvut.cz/ontologies/2009/failures.owl>.

integrity constraint set. Most of the time the clash was caused by inserting invalid data, like failure records without a link to corresponding structure records mentioned above. In these cases civil engineering experts were notified about the integrity constraints violation immediately while authoring the records in Protégé and thus they were able to repair them before saving the ontology (and thus before providing the invalid data to the StruFail system business logic). Comparing to the traditional design (see above), this immediate feedback caused significant time savings for the application developers as they didn't need to debug the StruFail business logic that would crash in the application runtime otherwise.

A few times civil engineering experts identified that the clash was caused by an incorrect assumption of the domain knowledge, and thus incorrect definition of integrity constraints. For example, the original requirement for a failure record was to have exactly one factor attached, which corresponds to the integrity constraint $\text{Failure} \sqsubseteq (= 1 \text{ hasFactor})$. This integrity constraint was violated by several failure records that have more than one factor attached. In these cases, it was necessary to adjust the integrity constraints set, regenerate the object model, adjust business logic and recompile the application.

When civil engineering experts filled failure records using Protégé, some of the records were missing a link `isFailureOf` to the structure record they referred to, which caused violation of the integrity constraint (5.2). As a result, when accessing the ontology, JOPA generated an integrity constraints violation exception that provided the information about invalid records of failures. After interpreting this integrity constraint violation to the civil engineering experts, they repaired each failure record by linking it to the corresponding structure record and the ontology became compliant with StruFail again.

As most of the user interactions with the system are read-only and the ontology does not change, the transactional support in JOPA saved many consistency checks during retrieval of failure records by StruFail users. The reason is that queries could be evaluated (see activities FR2 to FS3 in Figure 4.1) without a consistency check. Once a change in the data is detected (FR5), e.g. a user edits a failure record, evaluating a new query is not possible until the failure edits are committed or rolled back and a new transaction is started.

6.1.2. System Usage

Let's briefly explore the capabilities of StruFail. The web UI allows users to both (i) search various information about known failure cases, (ii) report new failures, the latter being available only for registered users.

Exploration of Known Failure Cases

There are several options for retrieving information from StruFail. Menu items *Structure* and *Failure* show a page with structure/failure lists. Selection of the particular structure/failure in the list navigates the user to its details.

A more advanced querying is available in the menu item *Explore*. Here, seven complex query templates are defined that can be instantiated by the user. The queries can

be parameterized by ontological classes and properties forming hierarchies (through \sqsubseteq axioms). This allows posing both general and specific queries to cover large scale of user requirements. One of the example queries is shown (together with its results) in Figure 6.4.

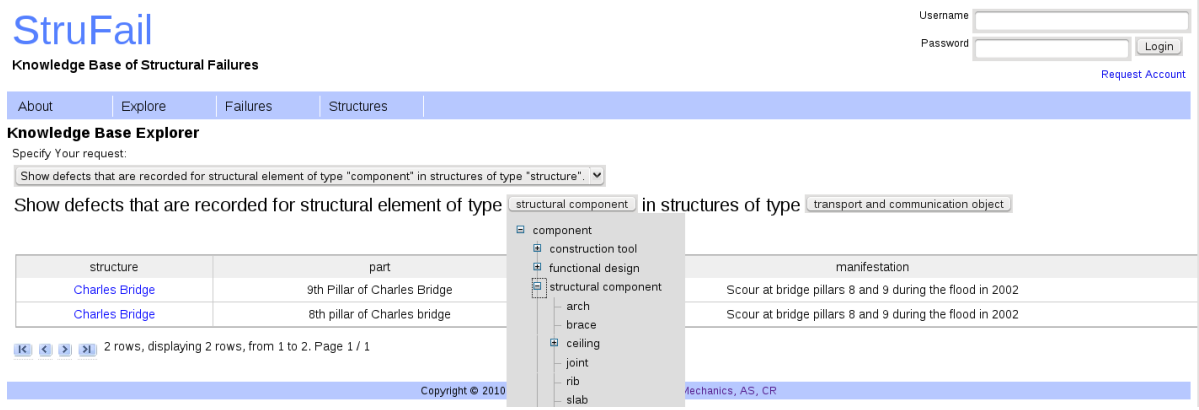


Figure 6.4.: Exploration of the knowledge base by predefined queries.

This query retrieves all manifestations of failures that occur on a specified component (e.g. “structural component” in this case) of a specified structure (e.g. “transport and communication object” in this case). The query result is a table, each row of which displays one particular manifestation of one particular component of one particular structure. In this case two rows are retrieved that represent a scour manifested on the 8th and 9th pillar (components) of the Charles Bridge. The query is a conjunctive ABox query of the following form

$$[PV(?m, hasAffectedPart, ?c), Ty(?c, C_c), PV(!f, hasManifestation, ?m), \\ PV(!f, isFailureOf, ?s), Ty(?s, C_s)],$$

where $?m$, $?c$, $!f$, $?s$ are variables denoting respectively a manifestation, component, failure and structure and C_c , resp. C_s represent the actual parameter selected by the user in the drop-down tree component, as depicted in Figure 6.4 (in this case **StructuralComponent** and **TransportAndCommunicationObject**). Only the failure can be represented by an undistinguished variable $!f$ as it does not appear in the result set.

In addition to conjunctive ABox queries, the query set also contains expressive SPARQL-DL queries, evaluated by the SPARQL-DL^{NOT} query engine introduced in Section 4.2.2. For example, the following query asks for defects (and their manifestations) of structures of given type, together with the particular type:

$$[Ty(?s, ?st), SCO(?st, C_s), PV(?f, isFailureOf, ?s), PV(?f, hasManifestation, ?m)],$$

where $?s$, $?st$, $?f$, $?m$ are variables denoting respectively a structure, its type, related failure and its manifestation. Another example query asks for a structure with its

[About](#)
[Explore](#)
[Failures](#)
[Structures](#)
[Add](#)

Edit Failure

Failure of structure [Edit structure](#)
[Create new structure](#)

Short description of failure

Cause description

Remedial works
[Add measure](#)

Defects caused by the failure


Defects observed on structure

Type of manifestation [ADD](#)

Affected part(s) [ADD](#)

[Add defec](#)

Edit attachn

 Remediation work at pillars 8 and 9

[Add resource](#)

short description

[Browse...](#) [Upload](#)

[Save changes](#)

Copyright © 2010 Institute of Theoretical

Figure 6.5.: Failure report.

type that is *related* to an event. The type of relationship can be different (defined by subproperties *isFailureOf* and *isRepairOf* of OWL object property *affects*) and is used to distinguish failures and repairs related to a structure:

$$[PV(?s, ?iEO, ?e), SPO(?iEO, affects), Ty(?s, ?st),$$

where $?s$, $?st$, $?iEO$, $?e$ are variables denoting respectively a structure, its type, the type of relationship (*isFailureOf*, *isRepairOf*) and the particular failure/repair.

These queries provide the user with a completely new experience when interacting with the ontological knowledge due to the possibility of retrieving object types (OWL classes) and relationship types (OWL properties) along with the objects themselves.

Failure case reporting

Ontological knowledge is used not only for querying the knowledge base, but also for inputting new failure records. Upon registration the new user is verified by the database administrator and is given authorization to add new failure cases to the knowledge base. New failures can be reported after logging into the web application using the menu *Failure* \rightarrow *Add Failure* (see example of a filled failure report form in Figure 6.5).

When filling-in a failure report a special attention should be paid to the categorization of measures, manifestations and affected parts. The reporter is encouraged to classify

these attributes to predefined categories (e.g. **Pillar** as a subtype of a component, as shown in the Figure 6.5). These categories can be chosen from class hierarchies that correspond to the inferred taxonomies for the respective element (component, structure, manifestation, etc.). Both structures and failures can be equipped with photos/other related resources, like documents, measurement reports, etc.

7. Conclusions

This thesis proposes a methodology and a framework for designing information systems on top of OWL ontologies, including expressive query language SPARQL-DL^{NOT} and its evaluation and optimization techniques. The framework allows the application to insert/change/remove/retrieve data in evolving domain ontologies while the ontology is “compliant” with the application object model. The notion of compliance is materialized by integrity constraints that have to be valid on all ontological data before they can be modified/retrieved by the application. This formalization of the application-ontology compliance are a clear added value comparing to the state-of-the-art object-ontology mappings that are ad-hoc. In my future work I would like to elaborate other types of integrity constraints based on OWL 2 DL features, including inverse roles and OWL 2 keys. Also, other means for persisting OWL 2 will be explored including RDF triple-stores or native OWL storage mechanisms, like Pellet DB [125].

An important part of the proposed framework is the expressive query language SPARQL-DL^{NOT}, its evaluation and optimization techniques that are presented and efficiency of which is demonstrated on various experiments. The optimizations focus on queries with undistinguished variables that is one of the crucial distinction of SPARQL-DL^{NOT} comparing to other expressive query languages for OWL. Although introduced core optimization technique makes evaluation of queries with undistinguished variables significantly faster, still, during the implementation of these optimizations in the Pellet reasoner it turned out that evaluation of cores remains the major bottleneck for queries with undistinguished variables. Thus, next work will study the overall impact of different core evaluation strategies to avoid as many instance checks as possible.

Other optimizations involve dynamic query reordering targeted on long queries as well as pruning variable binding using taxonomies for variables in class and property positions. The presented algorithms were implemented and tested on top of a tableau algorithm for OWL 2 DL consistency checking in the Pellet reasoner and the OWL2Query engine. Still, its integration with another tableau-based inference engine, like [64], to handle cycles of undistinguished variables seems beneficial and will be studied in detail in the future. For *SHIN*, the presented techniques could be complemented by ABox summarization [126] to avoid full computation of sets of completion trees for efficient evaluation of queries without undistinguished variables. SPARQL-DL^{NOT} query engine performance is measured on well-known artificial benchmark data sets LUBM and UOB.

A prototype implementation JOPA of the introduced methodology was used during the design of two applications: StruFail – a system for description of structural failures, and SHM CMS – a system for semantic content management. The StruFail system prototype has been used for demonstration of the application of the methodology in this thesis. Although the sponsoring project CHEF ended before the large-scale deployment

of the StruFail system could have taken place, the system has been positively accepted by several domain experts in the community. Based on their interest, the experience gained during StruFail design and development is currently used for design and development of a more sophisticated structural failure knowledge-based information system¹ within a 5-year grant of the Czech Ministry of Culture. The scope of the novel system is broader. It will enhance the features of the StruFail prototype e.g. with advanced failure authoring tool as well as advanced search options, including full-text search capability as well as definition of more predefined query types and even a graphical query editor to allow creating arbitrary user-defined queries.

¹see <http://www.mondis.cz>, cit. 12/1/2012

Bibliography²

- [1] Thanh Tran, Peter Haase, Holger Lewen, Óscar Muñoz-García, Asunción Gómez-Pérez, and Rudi Studer. Lifecycle-support in architectures for ontology-based information systems. In *The Semantic Web, 6th International Semantic Web Conference, 2nd Asian Semantic Web Conference (ISWC'07/ASWC'07)*, pages 508–522, Berlin, Heidelberg, 2007. Springer-Verlag.
- [2] Graeme Stevenson and Simon Dobson. Sapphire: Generating Java Runtime Artefacts from OWL Ontologies. In *Proceedings of the Third International Workshop on Ontology-Driven Information Systems Engineering*, 2011.
- [3] Jiao Tao, Evren Sirin, Jie Bao, and Deborah L. McGuinness. Integrity Constraints in OWL. In *Proceedings of the Twenty-Fourth AAAI Conference on Artificial Intelligence*, 2010.
- [4] Thomas R. Gruber. A Translation Approach to Portable Ontology Specification. *Knowledge Acquisition*, 5(2):199–220, 1993.
- [5] Monika Žáková, Petr Křemen, Filip Železný, and Nada Lavrac. Automating Knowledge Discovery Workflow Composition Through Ontology-Based Planning. *IEEE Transactions on Automation Science and Engineering*, 8(2):253–264, 2011.
- [6] Lin Wang and Zhao Hongshuai. Ontology for Communication in Distributed Multi-agent System. *International Symposium on Distributed Computing and Applications to Business, Engineering and Science*, 0:588–592, 2010.
- [7] Tim Berners-Lee, James Hendler, and Ora Lassila. The Semantic Web. *Scientific American*, 284(5):34–43, 2001.
- [8] Nigel Shadbolt, Tim Berners-Lee, and Wendy Hall. The Semantic Web Revisited. *IEEE Intelligent Systems*, 21(3):96–101, 2006.
- [9] Asuncion Gomez-Perez, Mariano Fernandez-Lopez, and Oscar Corcho. *Ontological Engineering*. Springer, 2005.
- [10] Vinay K. Chaudhri, Adam Farquhar, Richard Fikes, Peter D. Karp, and James P. Rice. OKBC: A Programmatic Foundation for Knowledge Base Interoperability. In *Proceedings of the Fifteenth National Conference on Artificial Intelligence (AAAI-98)*, Madison, WI, 1998.

²Some bibliography items are not marked [online] although their URL is provided. These items are primarily published in print, but their online source URL is shown here for the comfort of the reader.

- [11] Jack Park and Sam Hunting, editors. *XML Topic Maps: Creating and Using Topic Maps for the Web*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2002.
- [12] Philippe Martin. Knowledge Represenatation in CGLF, CGIF, KIF, Frame-CG and Formalized English. In *Conceptual Structures: Integration and Interfaces, 10th International Conference on Conceptual Structures (UCCS'02)*, volume 2393 of *LNCS*, pages 77–91. Springer Verlag, 2002.
- [13] John F. Sowa. Semantics of Conceptual Graphs. In *Proceedings of the 17th conference on Association for Computational Linguistics*, pages 39–44. Association for Computational Linguistics, 1979.
- [14] Ian Horrocks, Bijan Parsia, Peter Patel-Schneider, and James Hendler. Semantic Web Architecture: Stack or Two Towers? In *Principles and Practice of Semantic Web Reasoning*, pages 37–41, 2005.
- [15] Jeremy J. Carroll and Graham Klyne. Resource Description Framework (RDF): Concepts and Abstract Syntax [online]. W3C Recommendation, W3C, 2004. Available at <http://www.w3.org/TR/2004/REC-rdf-concepts-20040210>, cit. 11/1/2012.
- [16] Ramanathan V. Guha and Dan Brickley. RDF Vocabulary Description Language 1.0: RDF Schema [online]. W3C Recommendation, W3C, 2004. Available at <http://www.w3.org/TR/2004/REC-rdf-schema-20040210>, cit. 11/1/2012.
- [17] Peter F. Patel-Schneider, Patrick Hayes, and Ian Horrocks. OWL Web Ontology Language Semantics and Abstract Syntax [online]. W3C Recommendation, W3C, 2004. Available at <http://www.w3.org/TR/2004/REC-owl-semantics-20040210>, cit. 11/1/2012.
- [18] Boris Motik, Peter F. Patel-Schneider, and Bijan Parsia. OWL 2 Web Ontology Language Structural Specification and Functional-Style Syntax [online]. W3C Recommendation, W3C, 2009. Available at <http://www.w3.org/TR/2009/REC-owl2-syntax-20091027>, cit. 11/1/2012.
- [19] Protégé Homepage [online]. Available at <http://protege.stanford.edu>, cit. 10/10/2011.
- [20] Michael Stonebraker and Gerald Held. Networks, Hierarchies and Relations in Data Base Management Systems. In *ACM Pacific*, pages 1–9, 1975.
- [21] Won Kim. *Introduction to object-oriented databases*. MIT Press, Cambridge, Mass., 1990.
- [22] Edgar F. Codd. A relational model of data for large shared data banks. *Communications of the ACM*, 13(6):377–387, 1970.

- [23] Asad M. Khattak, Khalid Latif, Songyoung Lee, and Young-Koo Lee. Ontology Evolution: A Survey and Future Challenges. In Dominik Ślęzak, Tai hoon Kim, Jianhua Ma, Wai-Chi Fang, Frode E. Sandnes, Byeong-Ho Kang, and Bongen Gu, editors, *U- and E-Service, Science and Technology*, volume 62, chapter 11, pages 68–75. Springer Berlin Heidelberg, Berlin, Heidelberg, 2009.
- [24] Eyal Oren, Benjamin Heitmann, and Stefan Decker. ActiveRDF: Embedding SemanticWeb Data into Object-oriented Languages. *Journal of Web Semantics*, 6(3), 2011.
- [25] Yinglin Wang, Xijuan Liu, and Rongwei Ye. Ontology Evolution Issues in Adaptable Information Management Systems. In *Proceedings of the 2008 IEEE International Conference on e-Business Engineering*, volume 0, pages 753–758, Los Alamitos, CA, USA, 2008. IEEE Computer Society.
- [26] Alexander Maedche, Boris Motik, Ljiljana Stojanovic, Rudi Studer, and Raphael Volz. Ontologies for Enterprise Knowledge Management. *Intelligent Systems, IEEE*, 18(2):26–33, 2003.
- [27] Vadim Ermolayev, Natalya Keberle, and Wolf-Ekkehard Matzke. An Upper Level Ontological Model for Engineering Design Performance Domain. In *Proceedings of the 27th International Conference on Conceptual Modeling (ER 2008)*, volume 5231 of *LNCS*, pages 98–113. Springer-Verlag, Berlin, Heidelberg, 2008.
- [28] Petr Křemen and Zdeněk Kouba. Ontology-Driven Information System Design. *IEEE Transactions on Systems, Man, and Cybernetics, Part C: Applications and Reviews*, to appear in 2012. Available at http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=6011704, cit. 11/1/2012.
- [29] Petr Křemen and Evren Sirin. SPARQL-DL Implementation Experience. In *Proceedings of OWL: Experiences and Directions (OWLED 2008)*, volume 496 of *CEUR*, 2008.
- [30] Petr Křemen and Zdeněk Kouba. Conjunctive Query Optimization in OWL2-DL. In *Proceedings of the 22th International Conference on Database and Expert System Applications (DEXA 2011)*, volume 6861 of *LNCS*. Springer Verlag, 2011.
- [31] Evren Sirin and Bijan Parsia. SPARQL-DL: SPARQL Query for OWL-DL. In *Proceedings of OWL: Experiences and Directions (OWLED 2007)*, volume 258 of *CEUR*, 2007.
- [32] Evren Sirin, Bijan Parsia, Bernardo C. Grau, Aditya Kalyanpur, and Yarden Katz. Pellet: A Practical OWL-DL Reasoner. *Journal of Web Semantics*, 5(2):51–53, 2007.

- [33] Petr Křemen, Marek Šmíd, and Zdeněk Kouba. OWLDiff: A Practical Tool for Comparison and Merge of OWL Ontologies. In *Proceedings of DEXA 2011 Workshops*, pages 229–233, Los Alamitos, CA, USA, 2011. IEEE Computer Society.
- [34] Miloš Drdácý, Jaroslav Valach, Petr Křemen, and Jan Abrahamčík. *Damage Database*, pages 185–195. Cultural Heritage Protection Against Flooding. Institute of Theoretical and Applied Mechanics, 2011.
- [35] Petr Křemen and Zdeněk Kouba. Incremental Approach to Error Explanations in Ontologies. In Klaus Tochtermann, Tassilo Pellegrini, and Sebastian Schaffert, editors, *Networked Knowledge - Networked Media, Integrating Knowledge Management, New Media Technologies and Semantic Systems*, Studies in Computational Intelligence. Springer, 2008.
- [36] Petr Křemen, Miroslav Blaško, and Zdeněk Kouba. *Semantic Annotation of Objects*, chapter XI. Handbook of Research on Social Dimensions of Semantic Technologies and Web Services. IGI Global, 2009.
- [37] Ian Horrocks, Peter F. Patel-Schneider, Harold Boley, Said Tabet, Benjamin Grosz, and Mike Dean. SWRL: A Semantic Web Rule Language Combining OWL and RuleML. W3C Member Submission, W3C, 2004.
- [38] Francesco M. Donini, Maurizio Lenzerini, Daniele Nardi, and Andrea Schaerf. AL-log: Integrating Datalog and Description Logics. *Journal of Intelligent Systems*, 10:227–252, 1998.
- [39] Alon Y. Levy and Marie-Christine Rousset. CARIN: A representation language combining Horn rules and description logics. In *ECAI*, pages 323–327. John Wiley and Sons, Chichester, 1996.
- [40] Riccardo Rosati. On the Decidability and Complexity of Integrating Ontologies and Rules. *Journal of Web Semantics*, 3(1):61–73, 2005.
- [41] Ian Horrocks, Peter F. Patel-Schneider, Sean Bechhofer, and Dmitry Tsarkov. OWL Rules: A Proposal and Prototype Implementation. *Journal of Web Semantics*, 3:723–731, 2005.
- [42] Patrick Hayes. RDF Semantics [online]. W3C Recommendation, W3C, 2004. Available at <http://www.w3.org/TR/rdf-mt>, cit. 11/1/2012.
- [43] Ian Horrocks. OWL: A Description Logic Based Ontology Language. In *Principles and Practice of Constraint Programming (CP 2005)*, pages 5–8, 2005.
- [44] Evgeny Zolin. Description Logic Complexity Navigator [online]. Available at <http://www.cs.man.ac.uk/~ezolin/dl>, cit. 11/9/2011.

- [45] Eric Prud'hommeaux and Andy Seaborne. SPARQL Query Language for RDF [online]. W3C Recommendation, W3C, 2008. Available at <http://www.w3.org/TR/rdf-sparql-query>, cit. 10/10/2011.
- [46] Steve Harris and Andy Seaborne. SPARQL 1.1 Query Language [online]. W3C Working Draft, W3C, 2012. Available at <http://www.w3.org/TR/2012/WD-sparql11-query-20120105>, cit. 13/1/2012.
- [47] Bijan Parsia. Querying the Web with SPARQL. In *Reasoning Web*, volume 4126 of *LNCS*, pages 53–67. Springer, 2006.
- [48] Franz Baader, Diego Calvanese, Deborah L. McGuinness, Daniele Nardi, and Peter Patel-Schneider, editors. *The Description Logic Handbook, Theory, Implementation and Applications*. Cambridge, 2003.
- [49] Ian Horrocks, Oliver Kutz, and Ulrike Sattler. The Even More Irresistible *SR*OTQ. In Patrick Doherty, John Mylopoulos, and Christopher A. Welty, editors, *Proceedings of the 10th International Conference on Principles of Knowledge Representation and Reasoning (KR2006)*, pages 57–67. AAAI Press, 2006.
- [50] Manfred Schmidt-Schauss and Gert Smolka. Attributive Concept Descriptions with Complements. *Artificial Intelligence*, 48(1):1–26, 1991.
- [51] Dmitry Tsarkov and Ian Horrocks. DL Reasoner vs. First-Order Prover. In *Proceedings of the 2003 Description Logic Workshop (DL 2003)*, volume 81 of *CEUR*, pages 152–159, 2003.
- [52] Dmitry Tsarkov, Alexandre Riazanov, Sean Bechhofer, and Ian Horrocks. Using Vampire to Reason with OWL. In *Proceedings of the International Semantic Web Conference (ISWC 2004)*, pages 471–485. Springer, 2004.
- [53] Boris Motik. *Reasoning in Description Logics using Resolution and Deductive Databases*. PhD thesis, Universitat Karlsruhe, 2006.
- [54] Hashim Habiballa. Resolution Based Reasoning in Description Logics. In *Proceedings of Znalosti 2006*, 2006.
- [55] Pellet Homepage [online]. Available at <http://pellet.owldl.com>, cit. 15/1/2007.
- [56] Ian Horrocks and Peter F. Patel-Schneider. Reducing OWL entailment to description logic satisfiability. *Journal of Web Semantics*, 1(4):345–357, 2004.
- [57] Matthew Horridge and Sean Bechhofer. The OWL API: A Java API for OWL ontologies. *Semantic Web – Interoperability, Usability, Applicability*, 2(1):11–21, 2011.
- [58] Jeremy J. Carroll, Ian Dickinson, Chris Dollin, Dave Reynolds, Andy Seaborne, and Kevin Wilkinson. Jena: implementing the semantic web recommendations. In *Proceedings of WWW (Alternate Track Papers & Posters)*, pages 74–83, 2004.

- [59] Thorsten Liebig, Marko Luther, Olaf Noppens, and Michael Wessel. OWLlink. *Semantic Web – Interoperability, Usability, Applicability*, 2(1):23–32, 2011.
- [60] JFact Homepage [online]. Available at <http://jfact.sourceforge.net>, cit. 12/11/2011.
- [61] HermiT homepage [online]. Available at <http://hermit-reasoner.com>, cit. 12/11/2011.
- [62] OWL 2 Reasoner Implementations [online]. Available at <http://www.w3.org/2007/OWL/wiki/Implementations>, cit. 1/12/2011.
- [63] Volker Haarslev and Ralf Möller. On the Scalability of Description Logic Instance Retrieval. *Journal of Automated Reasoning*, 41(2):99–142, 2008.
- [64] Magdalena Ortiz, Diego Calvanese, and Thomas Eiter. Data Complexity of Query Answering in Expressive Description Logics via Tableaux. *Journal of Automated Reasoning*, 41(1):61–98, 2008.
- [65] Martin J. O’Connor and Amar K. Das. SQWRL: a Query Language for OWL. In *Proceedings of OWL: Experiences and Directions (OWLED)*, volume 529 of *CEUR*, 2009.
- [66] Alexander Kubias, Simon Schenk, Steffen Staab, and Jeff Z. Pan. OWL SAIQL - An OWL DL Query Language for Ontology Extraction. In *Proceedings of OWL: Experiences and Directions (OWLED 2007)*, volume 258 of *CEUR*, 2007.
- [67] Evren Sirin, Blazej Bulka, and Michael Smith. Terp: Syntax for OWL-friendly SPARQL queries. In *Proceedings of OWL: Experiences and Directions (OWLED 2010)*, volume 614 of *CEUR*, 2010.
- [68] SPARQL-DL with OWL Functional Syntax [online]. Available at <http://code.google.com/p/twouse/wiki/SPARQLAS>, cit. 22/10/2011.
- [69] Yuanbo Guo, Zhengxiang Pan, and Jeff Heflin. LUBM: A benchmark for OWL knowledge base systems. *Journal of Web Semantics*, 3(2-3):158–182, 2005.
- [70] Ian Horrocks and Sergio Tessaris. A Conjunctive Query Language for Description Logic ABoxes. In *Proceedings of the Seventeenth National Conference on Artificial Intelligence and Twelfth Conference on Innovative Applications of Artificial Intelligence*, pages 399–404, 2000.
- [71] Evren Sirin and Bijan Parsia. Optimizations for Answering Conjunctive ABox Queries. In *Description Logics*, volume 189 of *CEUR*, 2006.
- [72] Ian Horrocks, Ulrike Sattler, and Stephan Tobies. Reasoning with Individuals for the Description Logic *SHIQ*. In *Proceedings of the 17th International Conference on Automated Deduction, CADE-17*, pages 482–496, London, UK, 2000. Springer-Verlag.

- [73] Birte Glimm, Ian Horrocks, Carsten Lutz, and Uli Sattler. Conjunctive Query Answering in the Description Logic *SHIQ*. In *Proceedings of the 20th International Joint Conference on Artificial Intelligence (IJCAI 2007)*, 2007.
- [74] Birte Glimm, Ian Horrocks, and Ulrike Sattler. Conjunctive Query Entailment for *SHOQ*. In *Proceedings of the 2007 Description Logic Workshop (DL 2007)*, volume 250 of *CEUR*, pages 65–75, 2007.
- [75] Holger Knublauch, Phil Tetlow, Evan Wallace, and Daniel Oberle. A Semantic Web Primer for Object-Oriented Software Developers [online]. W3C Note, W3C, 2006. Available at <http://www.w3.org/TR/2006/NOTE-sw-oosd-primer-20060309>, cit. 11/1/2012.
- [76] NeON Toolkit Homepage [online]. Available at <http://neon-toolkit.org>, cit. 5/7/2011.
- [77] TopBraid Composer Homepage [online]. Available at http://www.topquadrant.com/products/TB_Composer.html, cit. 1/8/2010.
- [78] SWOOP Homepage [online]. Available at <http://code.google.com/p/swoop>, cit. 17/1/2009.
- [79] Sabin Corneliu Buraga, Liliana Cojocaru, and Ovidiu Cătălin Nichifor. Survey on Web Ontology Editing Tools. *PERIODICA POLITECHNIC, Transactions on AUTOMATIC CONTROL and COMPUTER SCIENCE*, 2006.
- [80] Seongwook Youn, Anchit Arora, Preetham Chandrasekhar, Paavany Jayanty, Ashish Mestry and Shikha Sethi. Survey about Ontology Development Tools for Ontology-based Knowledge Management [online]. Available at <http://www-scf.usc.edu/~csci586/projects/ontology-survey.doc>, cit. 25/11/2005.
- [81] Tim Finin, Yun Peng, R. Scott, Cost Joel, Sachs Anupam Joshi, Pavan Reddivari, Rong Pan, Vishal Doshi, and Li Ding. Swoogle: A Search and Metadata Engine for the Semantic Web. In *Proceedings of the Thirteenth ACM Conference on Information and Knowledge Management*, pages 652–659. ACM Press, 2004.
- [82] Donato Griesi, Maria Pazienza, and Armando Stellato. Semantic Turkey: A Semantic Bookmarking Tool (System Description). In Enrico Franconi, Michael Kifer, and Wolfgang May, editors, *The Semantic Web: Research and Applications*, volume 4519 of *LNCS*, pages 779–788, Berlin, Heidelberg, 2007. Springer-Verlag.
- [83] Mozilla Firefox Homepage [online]. Available at <http://www.mozilla.org/firefox>, cit. 10/9/2011.

- [84] Sebastian Schaffert. IkeWiki: A Semantic Wiki for Collaborative Knowledge Management. In *Proceedings of the 15th IEEE International Workshops on Enabling Technologies: Infrastructure for Collaborative Enterprises*, pages 388–396, Washington, DC, USA, 2006. IEEE Computer Society.
- [85] David Frankel. *Model Driven Architecture: Applying MDA to Enterprise Computing*. John Wiley & Sons, Inc., New York, NY, USA, 2002.
- [86] Lv an Tang, Hongyan Li, Baojun Qiu, Meimei Li, Jianjun Wang, Lei Wang, Bin Zhou, Dongqing Yang, and Shiwei Tang. WISE: A Prototype for Ontology Driven Development of Web Information Systems. In *APWeb*, pages 1163–1167, 2006.
- [87] Mauri Leppänen. A Perspective Ontology and IS Perspectives. In *Proceeding of the 2008 conference on Information Modelling and Knowledge Bases XIX*, pages 257–275, Amsterdam, The Netherlands, 2008. IOS Press.
- [88] Rosemary Shrestha, Hector Sanchez, Claudio Ayala, Peter Wenzl, and Arnaud Elizabeth. Ontology-driven International Maize Information System (IMIS) for Phenotypic and Genotypic Data Exchange, 2010. Available at <http://precedings.nature.com/documents/5029>, cit. 15/10/2011.
- [89] Amit Sheth and Cartic Ramakrishnan. Semantic (Web) Technology in Action: Ontology Driven Information Systems for Search, Integration and Analysis. *IEEE Data Engineering Bulletin*, 26:40–48, 2003.
- [90] Craig E. Kuziemsky and Francis Lau. A Four Stage Approach for Ontology-based Health Information System Design. *Artificial Intelligence In Medicine*, 50(3):133–148, 2010.
- [91] Moein Mehrolhassani and Atilla Elci. OLS: An Ontology Based Information System. In *Proceedings of International Conference on Complex, Intelligent and Software Intensive Systems (CISIS 2008)*, pages 892–898, 2008.
- [92] Gunnar A. Grimnes, Leo Sauermann, and Ansgar Bernardi. The Personal Knowledge Workbench of the NEPOMUK Semantic Desktop. In *Proceedings of the 6th European Semantic Web Conference on The Semantic Web: Research and Applications*, pages 836–840, 2009.
- [93] Max Völkel. RDFReactor – From Ontologies to Programatic Data Access. In *Proceedings of Jena User Conference*. HP Bristol, 2006.
- [94] Edward Kawas and Mark D. Wilkinson. OWL2Perl: creating Perl modules from OWL class definitions. *Bioinformatics*, 26(18):2357–8, 2010.
- [95] SurfRDF Homepage [online]. Available at <http://code.google.com/p/surfrdf>, cit. 10/8/2011.

- [96] RDFAPI Homepage [online]. Available at <http://www4.wiwiss.fu-berlin.de/bizer/rdfapi>, cit. 10/8/2011.
- [97] Jonas von Malottki. Java OWL APIs [online]. Available at <http://wiki.yoshtec.com/java-owl-api>, cit. 6/10/2010.
- [98] H. Story. Semantic Object (Metadata) Mapper [online]. Available at <http://sommer.dev.java.net/sommer>, cit. 6/10/2010.
- [99] Carsten Saathoff, Stefan Scheglmann, and Simon Schenk. Winter: Mapping RDF to POJOs revisited. In *Proceedings of the ESWC 2009 Demo and Poster Session*, 2009.
- [100] Peter Mika and James Leigh. Elmo User Guide [online]. Available at <http://www.openrdf.org/doc/elmo/1.5/user-guide.html>, cit. 6/10/2010.
- [101] RDF2Java Homepage [online]. Available at <http://rdf2java.opendfki.de>, cit. 21/8/2011.
- [102] Jeen Broekstra, Arjohn Kampman, and Frank van Harmelen. Sesame: A generic architecture for storing and querying RDF and RDF Schema. In *Proceedings of the first International Semantic Web Conference (ISWC 2002)*, volume 2342 of *LNCS*, pages 54–68. Springer Verlag, 2002.
- [103] Matthias Quasthoff and Christoph Meinel. Supporting Object-Oriented Programming of Semantic-Web Software. *IEEE Transactions on Systems, Man, and Cybernetics, Part C: Applications and Reviews*, 42(1):15–24, 2012.
- [104] Jürgen Frohn, Georg Lausen, and Heinz Uphoff. Access to Objects by Path Expressions and Rules. In *Proceedings of the 20th International Conference on Very Large Data Bases (VLDB'94)*, pages 273–284, 1994.
- [105] Michael Grove. Empire: RDF & SPARQL Meet JPA [online]. Available at <http://semanticweb.com/empire-rdf-sparql-meet-jpa.b15617>, cit. 11/8/2011.
- [106] Mike Keith and Merrick Schincariol. *Pro JPA 2: Mastering the Java Persistence API*. Apress, Berkely, CA, USA, 1st edition, 2009.
- [107] Michael Zimmermann. Owl2Java - A Java Code Generator for OWL [online]. Available at <http://www.incunabulum.de/projects/it/owl2java>, cit. 6/10/2010.
- [108] Jonas von Malottki. JAOb (Java Architecture for OWL Binding) [online]. Available at <http://wiki.yoshtec.com/jaob>, cit. 11/8/2011.
- [109] Benjamin H. Szekely and J. Betz. Jastor Homepage [online]. Available at <http://jastor.sourceforge.net>, cit. 6/10/2010.

- [110] Aditya Kalyanpur, Daniel J. Pastor, Steve Battle, and Julian A. Padget. Automatic Mapping of OWL Ontologies into Java. In *Proceedings of the Sixteenth International Conference on Software Engineering & Knowledge Engineering (SEKE'2004)*, pages 98–103, 2004.
- [111] Fernando S. Parreiras, Carsten Saathoff, Tobias Walter, Thomas Franz, and Steffen Staab. APIs à gogo: Automatic Generation of Ontology APIs. In *Proceedings of the 2009 IEEE International Conference on Semantic Computing (ICSC '09)*, pages 342–348, Washington, DC, USA, 2009. IEEE Computer Society.
- [112] Maxim Davidovsky, Vadim Ermolayev, and Vyacheslav Tolok. Instance Migration Between Ontologies having Structural Differences. *International Journal on Artificial Intelligence Tools. Topical Issue on Intelligent Distributed Systems*, 20(6):1127–1156, 2011.
- [113] Volker Haarslev, Ralf Möller, and Michael Wessel. Querying the Semantic Web with Racer + nRQL. In *Proceedings of the KI-2004 International Workshop on Applications of Description Logics (ADL'04)*, 2004.
- [114] Jim Gray and Andreas Reuter. *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1992.
- [115] Ramnivas Laddad. *AspectJ in Action: Practical Aspect-Oriented Programming*. Manning Publications Co., Greenwich, CT, USA, 2009.
- [116] Martin Ledvinka. Transactional Processing of Ontologies. Bachelor's thesis, Czech Technical University in Prague, 2011. Available at https://dip.felk.cvut.cz/browse/pdfcache/ledvima1_2011bach.pdf, cit. 14/10/2011.
- [117] OWLDB Homepage [online]. Available at <http://owldb.sourceforge.net>, cit. 22/4/2011.
- [118] Bogdan Kostov. Visualization of Expressive Queries into OWL 2 Ontologies. Master's thesis, Czech Technical University in Prague, 2010. Available at <http://cyber.felk.cvut.cz/research/theses/papers/124.pdf>, cit. 14/10/2011.
- [119] SPARQL-DL API [online]. Available at <http://www.derivo.de/en/resources/sparql-dl-api.html>, cit. 20/10/2011.
- [120] Li Ma, Yang Yang, Zhaoming Qiu, GuoTong Xie, Yue Pan, and Shengping Liu. Towards a Complete OWL Ontology Benchmark. In *ESWC*, pages 125–139, 2006.
- [121] Jan Abrahamčík. Using Ontologies for Knowledge Management of Structural Failures of Constructions. Master's thesis, Czech Technical University in Prague, 2010. Available at <http://cyber.felk.cvut.cz/research/theses/papers/121.pdf>, cit. 20/10/2010.

- [122] Kamil Matoušek and Zdeněk Kouba. ODCA - Ontology-Based Document and Content Annotation in Structural Health Monitoring. In *Workshop on Database and Expert Systems Applications*, volume 2, pages 302–305, Berlin, 2011. IEEE Computer Society.
- [123] Jiří Kopecký. Semantic Web-Based Content Management System. Bachelor's thesis, Czech Technical University in Prague, 2011. Available at https://dip.felk.cvut.cz/browse/pdfcache/kopecji5_2011bach.pdf, cit. 20/11/2011.
- [124] Glassfish Homepage [online]. Available at <http://glassfish.java.net>, cit. 1/8/2009.
- [125] PelletDB homepage [online]. Available at <http://clarkparsia.com/pelletdb>, cit. 22/4/2011.
- [126] Julian Dolby, Achille Fokoue, Aditya Kalyanpur, Li Ma, Edith Schonberg, Kavitha Srinivas, and Xingzhi Sun. Scalable Grounded Conjunctive Query Evaluation over Large and Expressive Knowledge Bases. In *Proceedings of the 7th International Conference on The Semantic Web*, volume 5318 of *LNCS*, pages 403–418, Berlin, Heidelberg, 2008. Springer-Verlag.
- [127] Matthew Horridge, Bijan Parsia, and Ulrike Sattler. Justification Oriented Proofs in OWL. In *Proceedings of the 9th International Semantic Web Conference (ISWC 2010)*, volume 6496 of *LNCS*, pages 354–369, 2010.
- [128] Petr Křemen and Zdeněk Kouba. Incremental Approach to Error Explanations in Ontologies. In *Proceedings of the 7th International Conference on Knowledge Management (I-KNOW 2007)*, pages 332–339. Know-Center, Austria, 2007.
- [129] Ludmila Tichá, Zdeňka Cívínová, Michaela Morysková, Ilona Trtíková, and Lenka Němečková. Jak psát vysokoškolské závěrečné práce [online], 2011. Available at <http://knihovna.cvut.cz/redirect.php?id.file=1017>, cit. 5/12/2011.

Acronyms

Notation	Description	Page List
DCQ^{NOT}	Distinguished Conjunctive Queries with Negation	28, 29, 40–42, 56, 57, 60–62, 67, 82
CWA	Closed World Assumption	2, 41, 45, 56
FOPL	First-Order Predicate Logic	14, 19
IRI	Internationalized Resource Identifier	80–82
JOPA	Java OWL Persistence API	77
JPA	Java Persistence API	51, 83
JPQL	Java Persistence Query Language	83
MDA	Model-Driven Architecture	48
NNF	Negation Normal Form	20, 22
OIS	Ontology-based Information System	28, 45, 46, 52, 54, 60
OOM	Object-Ontology Mapping	48, 51, 55
OWA	Open World Assumption	2, 17, 41, 45
OWL	Web Ontology Language	1–3, 7, 11, 12, 29, 45, 46, 48, 49, 52, 77
RDF	Resource Description Framework	1, 7, 8, 11–13, 29, 49
RDFS	RDF Schema	1, 7, 11, 45, 46, 49, 51, 52
SOA	Service Oriented Architecture	46
SPARQL	SPARQL Query Language for RDF	7, 12, 13, 29, 49
SWRL	Semantic Web Rule Language	7
UI	User Interface	46, 48, 95, 96
UML	Unified Modeling Language	52
URI	Uniform Resource Identifier	8, 9, 12, 13

A. SPARQL-DL Atom Abbreviations

full atom name	abbreviation	full atom name	abbreviation
Type	Ty	EquivalentProperty	EP
PropertyValue	PV	InverseOf	IO
SameAs	SA	*ObjectProperty	OP
DifferentFrom	DF	*DataProperty	DP
SubClassOf	SCO	Functional	Fun
EquivalentClass	EC	InverseFunctional	IFun
DisjointWith	DW	Transitive	Trans
ComplementOf	CO	Symmetric	Sym
SubPropertyOf	SPO	*Annotation	A

Table A.1.: SPARQL-DL atom names. Atoms prefixed with * sign are not considered in this thesis, as explained in Section 2.7.1.

B. SPARQL-DL^{NOT} Atom Cost Estimates

This section shows rough estimates of the cost of evaluating SPARQL-DL^{NOT} query atoms that can be used for computing the static query reordering optimization and dynamic query reordering optimization, as shown in Section 4.2.2. In the tables presented in this section, the following notion is used:

$\chi(o)$ denotes the maximal cost of the particular reasoner operation, as specified in Table 2.5,

$\chi^s(o_u)$, (**resp.** $\chi^s(o_b)$) denotes the cost of evaluating a unary (resp. binary) *ground* atom,

$\chi_1^m(o_b)$ (**resp.** $\chi_2^m(o_b)$) denotes the cost of evaluating a binary atom in which exactly the first (resp. second) argument is a variable,

$\omega(o_u)$ (**resp.** $\omega(o_b)$) denotes the number of classes/instances/properties, depending on the type of a unary (resp. binary) query atom,

$\varepsilon_1(o_u)$ (**resp.** $\varepsilon_1(o_b)$, **resp.** $\varepsilon_2(o_b)$) denotes the estimate of the number of bindings for the first argument of o_u (resp. first argument of o_b , resp. second argument of o_b). As an example, $\varepsilon_1(\text{Ty})$ is the estimated number of instances for an arbitrary, but fixed, concept. A cheap and rough estimate of this number is an *average number of instances of computed accross all named concepts*, obtained from precompletion.

$\varepsilon_1(o, x_2)$ (**resp.** $\varepsilon_1(o, x_2, x_3)$, **resp.** $\varepsilon_2(o, x_1)$) estimates the number of bindings for an atom o , given that its second argument (resp. second and third argument, resp. first argument) is known. As an example, consider $\varepsilon_2(\text{SCO}, A)$ that denotes the estimated number of concepts that subsume A . Cheap estimate of this number is the *number of different axioms of the form $A \sqsubseteq \bullet$* that appear in the ontology. Another example would be $\varepsilon_1(\text{PV}, R, a)$ that estimates number of individuals that are connected through role R with a . This estimate can be get by counting the respective edges in the precompletion graph.

oper. \mathbf{o}_u	$\chi^s(\mathbf{o}_u)$	oper. \mathbf{o}_b	$\chi^s(\mathbf{o}_b)$	$\chi_1^m(\mathbf{o}_b)$	$\chi_2^m(\mathbf{o}_b)$	$\omega(\mathbf{o}_b)$
Fun, IFun	$\chi(\text{ISUBC})$	Ty	$\chi(\text{IC})$	$\chi(\text{IR})$	$\chi(\text{CR})$	CN
Trans	$\chi(\text{ISUBC})$	SA, DF	$\chi(\text{IC})$	$\chi(\text{IR})$	$\chi(\text{IR})$	IN
Ref, IRef	$\chi(\text{ISUBC})$	SCO, DW	$\chi(\text{ISUBC})$	$\chi(\text{SUBC})$	$\chi(\text{SUPC})$	CN
Sym	$\chi(\text{ISEQP})$	EC, CO	$\chi(\text{ISEQC})$	$\chi(\text{EQC})$	$\chi(\text{EQC})$	CN
ASym	$\chi(\text{ISUBC})$	SPO	$\chi(\text{ISUBP})$	$\chi(\text{SUBP})$	$\chi(\text{SUPP})$	CN
		EP, IO	$\chi(\text{ISEQP})$	$\chi(\text{EQP})$	$\chi(\text{EQP})$	RN

query atom q	ESTC(q, B)	ESTB(q, B)
$\mathbf{o}_u(\hat{x})$	$\chi^s(\mathbf{o}_u)$	1
$\mathbf{o}_u(?v)$	$ RN \cdot \chi^s(\mathbf{o}_u)$	$\varepsilon_1(\mathbf{o}_u)$
$\mathbf{o}_b(\hat{x}_1, \hat{x}_2)$	$\chi^s(\mathbf{o}_b)$	1
$\mathbf{o}_b(?v_1, \mathbf{x}_2)$	$\chi_1^m(\mathbf{o}_b)$	$\varepsilon_1(\mathbf{o}_b, \mathbf{x}_2)$
$\mathbf{o}_b(?v_1, ?v_2), ?v_2 \in B$	$\chi_1^m(\mathbf{o}_b)$	$\varepsilon_1(\mathbf{o}_b)$
$\mathbf{o}_b(\mathbf{x}_1, ?v_2)$	$\chi_2^m(\mathbf{o}_b)$	$\varepsilon_2(\mathbf{o}_b, \mathbf{x}_1)$
$\mathbf{o}_b(?v_1, ?v_2), ?v_1 \in B$	$\chi_2^m(\mathbf{o}_b)$	$\varepsilon_2(\mathbf{o}_b)$
$\mathbf{o}_b(?v_1, ?v_2)$	$ \omega(\mathbf{o}_b) \cdot \chi_1^m(\mathbf{o}_b)$	$ \omega(\mathbf{o}_b) \cdot \varepsilon_1(\mathbf{o}_b)$
$\text{PV}(\hat{\mathbf{a}}_1, \hat{R}, \hat{\mathbf{a}}_3)$	$\chi(\text{IC})$	1
$\text{PV}(?v_1, R, \mathbf{a}_3)$	$\chi(\text{IR})$	$\varepsilon_1(\text{PV}, R, \mathbf{a}_3)$
$\text{PV}(?v_1, R, ?v_3), ?v_3 \in B$	$\chi(\text{IR})$	$\varepsilon_1(\text{PV}, R)$
$\text{PV}(?v_1, ?v_2, ?v_3), \{?v_2, ?v_3\} \subseteq B$	$\chi(\text{IR})$	$\varepsilon_1(\text{PV})$
$\text{PV}(\mathbf{a}_1, R, ?v_3)$	$\chi(\text{IR})$	$\varepsilon_1(\text{PV}, R^-, \mathbf{a}_1)$
$\text{PV}(?v_1, R, ?v_3), ?v_1 \in B$	$\chi(\text{IR})$	$\varepsilon_1(\text{PV}, R^-)$
$\text{PV}(?v_1, ?v_2, ?v_3), \{?v_2, ?v_1\} \subseteq B$	$\chi(\text{IR})$	$\varepsilon_1(\text{PV})$
$\text{PV}(\hat{\mathbf{a}}_1, ?v_2, \hat{\mathbf{a}}_3)$	$ RN \cdot \chi(\text{IC})$	$\varepsilon_2(\text{PV})$
$\text{PV}(?v_1, R, ?v_3)$	$ IN \cdot \chi(\text{IR})$	$ IN \cdot \varepsilon_1(\text{PV}, R)$
$\text{PV}(?v_1, ?v_2, ?v_3), ?v_2 \in B$	$ IN \cdot \chi(\text{IR})$	$ IN \cdot \varepsilon_1(\text{PV})$
$\text{PV}(?v_1, ?v_2, \hat{\mathbf{a}}_3)$, or $\text{PV}(\hat{\mathbf{a}}_1, ?v_2, ?v_3)$	$ RN \cdot \chi(\text{IR})$	$ RN \cdot \varepsilon_2(\text{PV})$
$\text{PV}(?v_1, ?v_2, ?v_3)$	$ RN \cdot IN \cdot \chi(\text{IR})$	$ RN \cdot IN \cdot \varepsilon_1(\text{PV})$
CORE_γ	$ V(\hat{\gamma}) \chi(\text{IR}) + IN ^{V(\hat{\gamma})} \chi(\text{IC})$	$\varepsilon_2(Ty)^{ V(\hat{\gamma}) }$

Table B.1.: Rough estimates of SPARQL-DL^{NOT} atom evaluation costs. Notation: $V(\hat{Q}) = V(Q) \setminus B$ is a set of all unbound (i.e. not in B) distinguished variables in Q . Additionally, terms with circumflex denote either a named individual/concept/role, or a variable from B . E.g. \hat{x} means either x (individual) or $x \in B$ (variable bound with unknown individual). Handling of NOT atoms, as well as detailed description of the notation and rationale behind these estimates are presented in Section 4.2.2.

C. Comparing Ontologies using OWLDiff

During the development of StruFail, the ontology evolved dynamically, as already mentioned in Section 6.1. Thus, upon refinements of the ontology, conflicts between subsequent ontology versions occurred and had to be handled. Without proper tool support, detection of differences, inconsistencies, or redundancies between different ontology versions is time-consuming or even practically impossible for large ontologies. To tackle this issue, I designed and lead the development team of OWLDiff¹ [33], an open-source practical and easy-to-use award-winning tool for ontology comparison and merging. OWLDiff is widely recognized by the community (more than 3500 downloads of the tool since its early release in 2008) due to its comparison options, visualization options and Subversion integration. The tool is available in three versions: as (i) a standalone application that is suitable for comparing two ontologies without requiring any other tools, (ii) a NeON toolkit plug-in², and as (iii) a Protégé plug-in. Main development currently focuses on the Protégé plug-in.

Here, I only shortly sketch relevant features of OWLDiff designed by myself, i.e. syntactic comparison, redundancy-based comparison and the merge option. Another, fully semantic, comparison option CEX for rather restricted subset of OWL was implemented by Marek Šmíd. Details on all comparison options, together with OWLDiff tutorial can be found in [33].

C.1. OWLDiff Comparison Options

OWLDiff compares two OWL 2 ontologies, an original one $\mathcal{K}_o = \{\alpha_1^o, \dots, \alpha_O^o\}$ with its updated version $\mathcal{K}_u = \{\alpha_1^u, \dots, \alpha_U^u\}$, where each α_i^{ou} is either an OWL 2 axiom or an import declaration or an ontology annotation.

In the following examples, I consider an original ontology $\mathcal{K}_{o1} = \{\alpha_1^{o1}, \alpha_2^{o1}, \alpha_3^{o1}\}$, and

¹see <http://krizik.felk.cvut.cz/km/owldiff>, cit. 12/10/2011

²I won the second prize at the NeON Plug-in Developer's Contest in 2008 with the NeON toolkit plug-in, see http://www.neon-toolkit.org/wiki/Winners_Announced, cit. 12/1/2012.

its two updated versions $\mathcal{K}_{u2} = \mathcal{K}_{o1} \cup \{\alpha_1^{u2}, \alpha_2^{u2}\}$, and $\mathcal{K}_{u3} = \{\alpha_1^{o1}, \alpha_2^{o1}, \alpha_1^{u3}\}$, where:

$$\begin{aligned} \alpha_1^{o1} &: \text{ObjectWithFailures} \equiv \text{Structure} \sqcap \exists \text{hasFailure} \cdot \top, \\ \alpha_2^{o1} &: \text{Structure} \sqsubseteq \text{Object}, \\ \alpha_3^{o1} &: \text{TownHouse} \sqsubseteq \text{Object}, \\ \alpha_1^{u2} &: \text{Repair} \sqsubseteq \exists \text{isRepairOf} \cdot \top, \\ \alpha_2^{u2} &: \top \sqsubseteq \forall \text{isRepairOf} \cdot \text{Structure}, \\ \alpha_1^{u3} &: \text{TownHouse} \sqsubseteq \text{Structure}. \end{aligned}$$

This simple example is used to demonstrate the comparison options mentioned in the next sections.

C.1.1. Syntactic Diff

The simplest way to compare two ontologies, is the *syntactic comparison*, i.e. comparison of two sets of axioms, import declarations and ontology annotations. i.e. $\text{DIFF}(\mathcal{K}_o, \mathcal{K}_u) = \{\alpha \in \mathcal{K}_o \mid \alpha \notin \mathcal{K}_u\}$, or $\text{DIFF}(\mathcal{K}_u, \mathcal{K}_o) = \{\alpha \in \mathcal{K}_u \mid \alpha \notin \mathcal{K}_o\}$

For cases when the updated ontology augments the original one with new definitions, syntactic comparison is often sufficient to decide which axiom should be included in the merged ontology without compromising its consistency nor change in semantics of the original ontology. The reason is that whenever no named terms (classes/properties/individuals) are reused from \mathcal{K}_o for the newly added axioms to \mathcal{K}_u , the newly added axioms typically do not have any semantic impact on named terms from \mathcal{K}_o (although they might have, e.g. in presence of nominals). The set $\text{DIFF}(\mathcal{K}_o, \mathcal{K}_u)$ is cheap to compute, as no OWL 2 reasoning is needed.

Example 27 *Having ontology \mathcal{K}_{o1} and \mathcal{K}_{u2} , it is easy to see that $\text{DIFF}(\mathcal{K}_{o1}, \mathcal{K}_{u2}) = \emptyset$ and $\text{DIFF}(\mathcal{K}_{u2}, \mathcal{K}_{o1}) = \{\alpha_1^{u2}, \alpha_2^{u2}\}$.*

C.1.2. Redundancies

A more sophisticated comparison option that reflects the semantic impact of the newly added axioms is the computation of semantically redundant axioms, i.e. axioms that can be inferred from the other ontology. Information about semantic redundancies is suitable for cases when \mathcal{K}_u contains new (or lacks existing) axioms about classes and properties already present in \mathcal{K}_o .

The syntactic diff $\text{DIFF}(\mathcal{K}_u, \mathcal{K}_o)$ is divided to

- $\text{INF}(\mathcal{K}_u, \mathcal{K}_o) = \{\alpha \in \text{DIFF}(\mathcal{K}_u, \mathcal{K}_o) \mid \mathcal{K}_o \models \alpha\}$, i.e. axioms that can be inferred from \mathcal{K}_u , and to
- $\text{REST}(\mathcal{K}_u, \mathcal{K}_o) = \text{DIFF}(\mathcal{K}_u, \mathcal{K}_o) \setminus \text{INF}(\mathcal{K}_u, \mathcal{K}_o)$, i.e. remaining axioms.

When computing the set $\text{INF}(\mathcal{K}_u, \mathcal{K}_o)$ of semantically redundant axioms w.r.t. \mathcal{K}_o , entailment by \mathcal{K}_o of each axiom from $\text{DIFF}(\mathcal{K}_u, \mathcal{K}_o)$ has to be checked. Thus, in the worst case when $\text{DIFF}(\mathcal{K}_u, \mathcal{K}_o) = \mathcal{K}_u$, computing $\text{INF}(\mathcal{K}_u, \mathcal{K}_o)$ requires the number of OWL entailment checks equal to the number of axioms in \mathcal{K}_u .

To decide, whether an axiom $\alpha \in \text{INF}(\mathcal{K}_u, \mathcal{K}_o)$ is redundant and thus needs to be deleted from the merged version of \mathcal{K}_u and \mathcal{K}_o , a set $E(\alpha) = \{e_1, \dots, e_A\}$ of justifications (a.k.a explanations) is provided to the user. Each justification $e_i \in E(\alpha)$ for entailment of α from \mathcal{K}_o is a minimal set $e_i = \{\gamma_{i,1}, \dots, \gamma_{i,B}\} \subseteq \mathcal{K}_o$, such that $e_i \models \alpha$ and there is no strict subset f_i of e_i for which $f_i \models \alpha$. Although computing all possible justifications for a redundancy of a single axiom might be expensive (up to exponential in the number of OWL entailment checks needed), on demand computation of a single explanation described in [127] requires only linear number of entailment checks in the number of axioms of \mathcal{K}_u . OWLDiff is equipped with an two OWL entailment justifications algorithms, that I proposed in [128]; however, their description is out of the scope of this thesis. For more information about OWL entailment justifications see [127].

Example 28 *Syntactic comparison of \mathcal{K}_{u3} and \mathcal{K}_{o1} , results in $\text{DIFF}(\mathcal{K}_{o1}, \mathcal{K}_{u3}) = \{\alpha_3^{o1}\}$. Thus, computing redundancies requires a single entailment check: $\mathcal{K}_{u3} \models \alpha_3^{o1}$. In our case, this entailment holds and thus $\text{INF}(\mathcal{K}_{o1}, \mathcal{K}_{u3}) = \{\alpha_3^{o1}\}$. This means that α_3^{o1} can be inferred from the updated ontology \mathcal{K}_{u3} and thus the ontology designer might not need to include it to the merged ontology. Justification of the entailment $\mathcal{K}_{u3} \models \alpha_3^{o1}$ provides the ontology designer with the information why the redundancy occurs. This situation is shown in Figure C.2.*

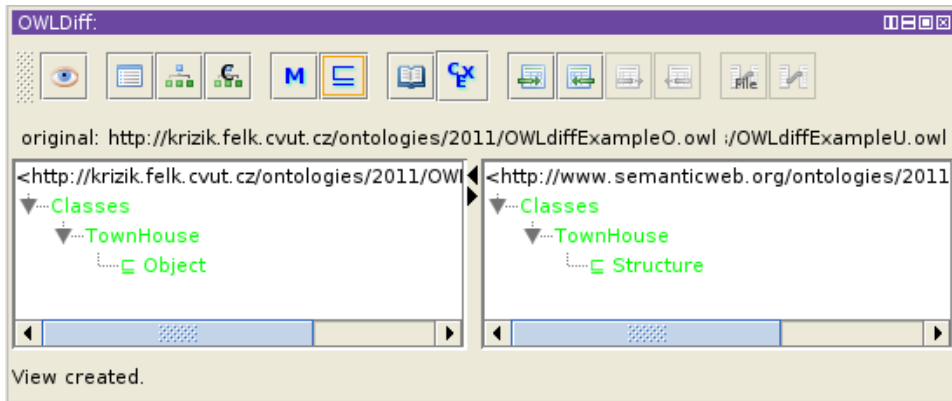


Figure C.1.: Protégé Syntactic comparison example. Axioms in one ontology, which do not appear in the other ontology, have green font color in its tree.

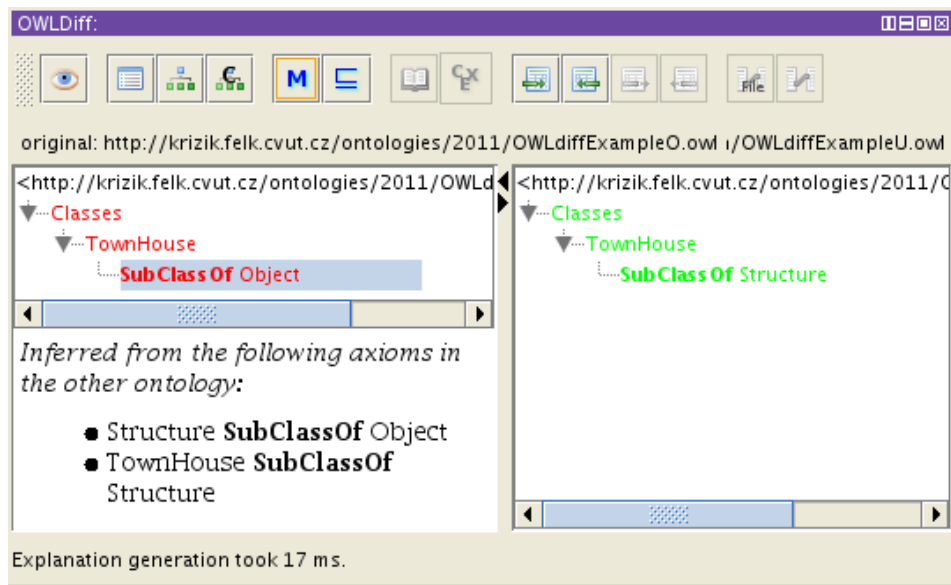


Figure C.2.: Redundancies comparison example. Axioms that can be inferred from the other ontology by an OWL reasoner, and thus might be redundant, are marked red. Upon selecting such an axiom, a justification of the inference appears in a box at the bottom of the tree.