

A Diagnosis and Repair Framework for *DL-Lite_A* Knowledge Bases

Michalis Chortis

Thesis submitted in partial fulfillment of the requirements for the
Masters' of Science degree in Computer Science

University of Crete
School of Sciences and Engineering
Computer Science Department
Voutes University Campus, Heraklion, GR-71003, Greece

Thesis Advisor: Prof. *Vassilis Christophides*

UNIVERSITY OF CRETE
COMPUTER SCIENCE DEPARTMENT

A Diagnosis and Repair Framework for *DL-Lite_A* Knowledge Bases

Thesis submitted by
Michalis Chortis
in partial fulfillment of the requirements for the
Masters' of Science degree in Computer Science

THESIS APPROVAL

Author: _____
Michalis Chortis

Committee approvals: _____
Vassilis Christophides
Professor, Thesis Supervisor

Dimitris Plexousakis
Professor, Committee Member

Giorgos Flouris
Researcher, Committee Member

Departmental approval: _____
Angelos Bilas
Professor, Director of Graduate Studies

Heraklion, June 2014

Abstract

Several logical formalisms have been proposed in the literature for expressing structural and semantic integrity constraints of Linked Open Data (LOD). Still, the data quality of the datasets published in the LOD Cloud needs to be improved, as published linked data often violate such constraints. This lack of consistency may jeopardise the value of applications consuming linked data in an automatic way.

A major challenge in this respect, is to provide to the curators of linked data knowledge bases (KBs), the tools that will help them in detecting the violations of integrity constraints and in resolving them, in order to render the knowledge base valid and improve its data quality.

In this work, we propose a novel, fully automatic framework for detecting violations of integrity constraints (*diagnosis*) in KBs, by executing the appropriate queries over the data, as well as for resolving those violations (*repair*), by removing invalid data from the KB. Our approach takes into consideration both explicit and inferred ontology knowledge, by relying on the ontology language *DL-Lite_A* for the expression of several useful types of logical constraints and for the detection of data that are inconsistent with those constraints, while maintaining good computational properties.

The framework that is proposed in this work is modular, allowing each component to be implemented in a manner independent to the other components. This way, we are able to implement our framework with using off-the-shelf, state-of-the-art tools for several features, such as reasoning, query execution, etc.

We have implemented and evaluated our framework, showing that it is scalable for large datasets and numbers of invalidities, which are exhibited in reality by reference linked datasets, such as DBpedia. The evaluation also shows that our framework can be used over already deployed knowledge bases, without any further reconfiguration.

Περίληψη

Στη βιβλιογραφία έχουν προταθεί αρκετοί λογικοί φορμαλισμοί με σκοπό την έκφραση δομικών και σημασιολογικών περιορισμών ακεραιότητας, στο πλαίσιο των Διασυνδεδεμένων Ανοιχτών Δεδομένων (Linked Open Data - LOD). Ωστόσο, η ανάγκη βελτίωσης της ποιότητας των δεδομένων που δημοσιεύονται στο σύννεφο Διασυνδεδεμένων Ανοιχτών Δεδομένων (LOD Cloud) παραμένει, καθώς τα δημοσιευμένα ανοιχτά δεδομένα συχνά παραβιάζουν τέτοιου είδους περιορισμούς ακεραιότητας. Αυτή η έλλειψη συνέπειας των δεδομένων με τους περιορισμούς, μπορεί να θέσει σε κίνδυνο την αξία εφαρμογών που καταναλώνουν ανοιχτά δεδομένα με αυτόματο τρόπο.

Μία βασική πρόκληση για τη βελτίωση της ποιότητας των δεδομένων, είναι η παροχή στους διαχειριστές των γνωσιακών βάσεων, εργαλείων που θα τους βοηθούν στον εντοπισμό παραβιάσεων των περιορισμών ακεραιότητας, καθώς και στην επίλυση τέτοιων παραβιάσεων.

Στην εργασία αυτή προτείνεται ένα νέο, πλήρως αυτόματοποιημένο πλαίσιο εντοπισμού παραβιάσεων περιορισμών ακεραιότητας σε γνωσιακές βάσεις, εκτελώντας τις απαραίτητες επερωτήσεις, καθώς και επιδιόρθωσης αυτών των παραβιάσεων, αφαιρώντας ασυνεπή δεδομένα από τη γνωσιακή βάση. Η μέθοδος που παρουσιάζεται, λαμβάνει υπόψη την οντολογική γνώση που είτε προκύπτει από ρητές δηλώσεις, είτε προκύπτει σε συμπέρασμα από συνδυασμό ρητών δηλώσεων, χρησιμοποιώντας τη γλώσσα οντολογιών *DL-Lite_A* για την έκφραση χρήσιμων λογικών περιορισμών, καθώς και για τον εντοπισμό δεδομένων που είναι ασυνεπή με αυτούς τους περιορισμούς, διατηρώντας, παράλληλα, καλές ιδιότητες υπολογιστικής πολυπλοκότητας.

Το πλαίσιο που προτείνεται αποτελείται από συστατικά μέρη που μπορούν να υλοποιηθούν ανεξάρτητα το ένα με το άλλο, δίνοντας έτσι τη δυνατότητα για τη χρησιμοποίηση έτοιμων, σύγχρονων, βελτιστοποιημένων εργαλείων για διάφορες λειτουργίες, όπως είναι η εκτέλεση επερωτήσεων.

Στα πλαίσια αυτής της εργασίας, παρουσιάζεται η υλοποίηση του πλαισίου, καθώς και η αξιολόγηση της επίδοσής του, από την οποία εξάγεται το συμπέρασμα ότι μπορεί να χρησιμοποιηθεί για μεγάλα σύνολα δεδομένων και για μεγάλους αριθμούς παραβιάσεων, που παρατηρούνται στην πραγματικότητα σε γνωστές γνωσιακές βάσεις αναφοράς, όπως η DBpedia. Από την αξιολόγηση εξάγεται, επίσης, το συμπέρασμα πως το πλαίσιο που παρουσιάζεται σε αυτή την εργασία μπορεί να χρησιμοποιηθεί πάνω από γνωσιακές βάσεις που είναι ήδη σε λειτουργία, χωρίς καμία επιπλέον παραμετροποίηση.

Acknowledgements

I would like to express my special gratitude to my supervisor, Professor Vassilis Christophides, for his belief in me from the first moment, for the chance to be part of the Information Systems Laboratory of FORTH-ICS and, most of all, for his invaluable guidance in these first steps I made in research. I would also like to give my special thanks to Giorgos Flouris, who was by my side, always giving me useful advises and directions, throughout the whole process of my work for this thesis. Moreover, I would like to thank Professor Dimitris Plexousakis for his will to take part in the examination committee of this thesis.

My special thanks also go to the Institute of Computer Science at FORTH for their support, through my scholarship for the whole process that ended up with this thesis, as well as for providing the necessary support in terms of knowledge and materials. Moreover, I would like to thank the State Scholarships Foundation (IKY) for the very helpful scholarship they provided me.

This thesis would not be possible without the support from my friends, who, despite the distance, were always by my side, encouraging me every step of the way. Special thanks go to Danai, who has always been there for me, helping me in every way possible.

Last, but not least, I express my gratitude to my parents, Argiro and Thomas, and my brother Vaggelis for giving me all of their support and all the necessary stepping stones for my studies and my life.

Contents

1	Introduction	3
1.1	Semantic Web and Linked Open Data	3
1.2	The need for repairing in the Semantic Web - Motivation	4
1.3	Contributions	5
1.4	Structure of the rest of the Thesis	6
2	Related work	7
2.1	Repairing and CQA in relational databases	7
2.2	Repairing and CQA in linked data	8
3	Preliminaries	11
3.1	Description Logics - $DL-Lite_{\mathcal{A}}$	11
3.1.1	The Description Logics family of languages	11
3.1.2	$DL-Lite$ family of DL languages	12
3.1.3	$DL-Lite_{\mathcal{R}}$, $DL-Lite_{\mathcal{F}}$ and $DL-Lite_{\mathcal{FR}}$ languages	13
3.1.4	$DL-Lite_{\mathcal{A}}$ language	14
3.2	FOL-Reducibility	15
3.3	$DL-Lite_{\mathcal{A}}$ as a language for the diagnosis of constraint violations	16
3.4	Linked Data technologies	17
3.4.1	Resource Description Framework (RDF)	17
3.4.2	Web Ontology Language (OWL)	18
3.4.3	SPARQL Protocol and RDF Query Language	19
4	Overview of the diagnosis and repair framework	21
4.1	Diagnosis	21
4.1.1	Diagnosis component	21
4.1.2	Diagnosis algorithm	23
4.2	Repairing	26
4.2.1	Repairing component	26
4.2.2	Repairing algorithm	26
5	Diagnosis and repair framework implementation	29
5.1	Used tools and libraries	29
5.1.1	OpenLink Virtuoso Open-Source Edition	29

5.1.2	Apache Jena	29
5.1.3	JUNG - The Java Universal Network/Graph Framework . .	30
5.1.4	Vaadin framework	31
5.2	Framework architecture	31
5.2.1	Input component	32
5.2.2	Diagnosis component	33
5.2.3	Repairing component	37
5.3	Implemented applications	38
5.3.1	Console application	38
5.3.2	Web application	39
6	Evaluation of the framework	45
6.1	Environment of experiments	45
6.2	Description of used ontologies	46
6.2.1	1 st -3 rd sets of experiments	46
6.2.2	4 th set of experiments	46
6.3	Synthetic data generator for diagnosis and repair	47
6.3.1	Overview	47
6.3.2	Triple pattern graph	47
6.3.3	Generation of invalidities	49
6.4	Description of used datasets	51
6.4.1	1 st -3 rd sets of experiments	51
6.4.2	4 th set of experiments	52
6.5	Scalability and performance	52
6.6	Conclusions from the evaluation	58
7	Conclusions and future work	61
7.1	Conclusions	61
7.2	Future work	61

List of Figures

1.1	W3C quality star scheme	3
1.2	The LOD cloud, as of September 2011	4
3.1	The graph view of the triple <code><example:John> a <example:Person></code>	18
4.1	The main features of the diagnosis and repair framework.	22
4.2	Example of an interdependency graph.	25
5.1	The architecture of the diagnosis and repair framework.	32
5.2	The input user interface of <i>OWLRepair</i>	40
5.3	The diagnosis user interface of <i>OWLRepair</i>	41
5.4	The diagnosis user interface of <i>OWLRepair</i>	42
5.5	The repair user interface of <i>OWLRepair</i>	42
5.6	The repair user interface of <i>OWLRepair</i>	43
6.1	Example of a triple pattern graph.	49
6.2	Performance for DBpedia ontology version 3.6 with datasets of varying dataset sizes and fixed number of invalid data assertions (10K triples).	54
6.3	Performance for DBpedia ontology version 3.9 with datasets of varying dataset sizes and fixed number of invalid data assertions (10K triples).	54
6.4	Performance for DBpedia ontology version 3.6 with datasets of varying number of invalid data assertions and fixed dataset size (10M triples).	56
6.5	Performance for DBpedia ontology version 3.9 with datasets of varying number of invalid data assertions and fixed dataset size (10M triples).	56
6.6	Comparison of our framework with the approach evaluated in [1].	57

List of Tables

4.1	Transformation of $DL-Lite_{\mathcal{A}}$ constraints to FOL queries	24
5.1	The input arguments of the console application	39
5.2	The output of the console application	39
6.1	Information on constraints contained in different DBpedia ontology versions.	46
6.2	Translation of $cln(\mathcal{T})$ constraints to vertices and edges of the triple pattern graph	50
6.3	Sizes (in triples) of the datasets used in the 4 th set of experiments .	52
6.4	Experiments performed on real datasets.	53
6.5	Comparison of the repairing deltas produced by using the greedy computation of the vertex cover and the 2-approximation, in the 4 th set of experiments	58

Chapter 1

Introduction

1.1 Semantic Web and Linked Open Data

In the last years, following the overall advance in computer science, a major new branch has been developed in various disciplines. This branch is concentrated in data exploration and analysis, trying to make use of the massive data flows coming from multiple and various sources. In line with this new situation, the Web has been evolving too. There has been a large initiative, mostly during the last decade, to provide, analyze and share data derived from all kinds of starting points (government, scientific research, social networks, enterprises etc.).

In order to obtain, analyze and share web data, it is extremely important to understand the underlying semantics of “things” expressed in this new world. In other words, one has to understand the meaning of things and the interlinking between them. In this context, the primary role is given to entities coming from diverse domains and sources (i.e. persons, places, institutions, abstract concepts etc.) and the relations between them.

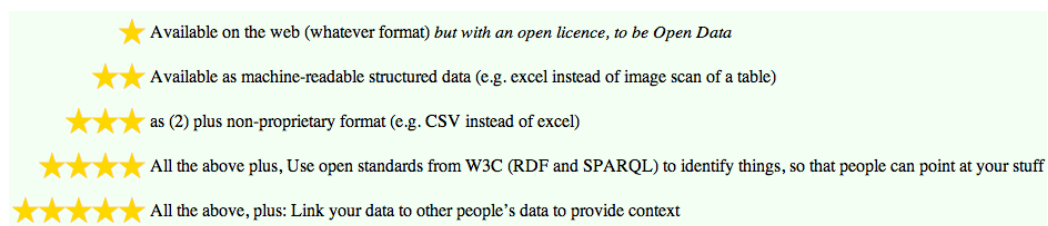


Figure 1.1: W3C quality star scheme

Due to the fact that new “players” continuously appear in the field of open data producing, editing and publishing, there is a great variety in the formats, quality and linking of the data available on the Web. Not all linked data is open and not all open data is linked. For that reason, W3C has walked some steps towards

the standardization of the Semantic Web and Linked Data¹. A star scheme for data quality assertion has been proposed (Figure 1.1) and there has been a major community effort to extend the Web with a data commons by publishing various open data sets as RDF on the Web and by setting RDF links between data items from different data sources². Figure 1.2 gives a good look on the extreme diversity of the information available as Linked Open Data (LOD). This figure presents the different sources that provide open data (graph nodes) and the available links between these sources (graph edges). An overview of the concept and technical principles of Linked Data is given in [2].

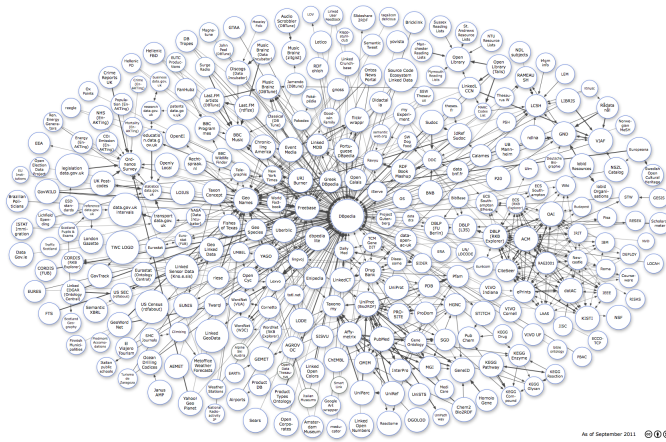


Figure 1.2: The LOD cloud, as of September 2011

1.2 The need for repairing in the Semantic Web - Motivation

Linked Open Data published on the Web of Data are often associated with various structural (e.g., primary key) and semantic (e.g., disjointness) integrity constraints. These constraints are usually expressed in ontology (e.g., in OWL [3, 4]) or database (e.g., embedded dependencies [5], used to prevent inconsistencies in the PROV data model³) logic frameworks. As a matter of fact, LOD data sources do not impose such constraints a priori, when data are created, so violations of integrity constraints must be detected and repaired a posteriori. As have been reported in [6], reference LOD sources, such as DBpedia⁴ or LinkedGeoData⁵, exhibit millions of

¹<http://www.w3.org/2001/sw/>

²<http://www.w3.org/wiki/LinkedData>

³<http://www.w3.org/TR/prov-constraints/>

⁴<http://dbpedia.org>

⁵<http://linkedgeo.org>

violations. In most of the cases, LOD are manually repaired by their curators or by their consuming applications, using, at best, diagnosis tools (such as the ones embedded in OWL reasoners) for detecting violations of various types of integrity constraints. Obviously, the manual repair of millions of violations is a time-consuming and error-prone task, a fact that seriously limits the data quality of the available LOD sources.

A major challenge in this respect is to detect violations of both structural and semantic integrity constraints when ontology reasoning is involved (i.e., violations of constraints like disjointness, functional constraints etc., taking into account logical inference and its interaction with those constraints). This challenge is the one that we aim to tackle in the context of this work.

1.3 Contributions

In this work, we propose a novel, automatic diagnosis and repairing framework for assisting curators in the arduous task of enforcing integrity constraints in large datasets. We provide an efficient methodology for detecting invalidities (*diagnosis*), as well as for automatically resolving them (*repairing*), in a manner that has minimal impact in terms of lost knowledge on the KB, according to the principles set out in earlier works [7, 8].

We consider detecting and repairing of invalidities attributed to constraints of a purely logical nature (e.g., concept disjointness). Constraints are expressed in the language *DL-Lite_A* [9], which belongs to the *DL-Lite* family of ontology languages that forms the foundation of *OWL 2 QL*⁶. The choice of *DL-Lite_A* was motivated by the fact that it is arguably rich enough to capture several useful types of integrity constraints and their interaction with implicit knowledge, while at the same time supporting efficient query answering [9].

The main contributions of our work are the following:

- We propose a framework for detecting and automatically repairing invalidities in datasets, for constraints that are expressed in *DL-Lite_A*, namely concept/property disjointness constraints, property domain/range disjointness constraints and functional constraints. Diagnosis of invalidities related to both explicit and inferred constraints can be performed in linear time with respect to the dataset size, whereas repairing can be performed in polynomial time with respect to the number of invalidities.
- We have implemented an operational repairing system for real-world applications. Our implementation is modular, allowing each component to be implemented in a manner independent to the other components. This way, we managed to reuse off-the-shelf, state-of-the-art tools for many of the components, such as reasoning, storage, query answering, etc.

⁶http://www.w3.org/TR/owl2-profiles/#OWL_2_QL

- We have experimentally evaluated the scalability and performance of our algorithms, using real and synthetic datasets. The main conclusion drawn is that our framework can scale for very large datasets, such as the one of DBpedia, as well as for large numbers (millions) of invalidities.

1.4 Structure of the rest of the Thesis

The rest of the Thesis is structured as follows:

- In Chapter 2, we sketch the work that has been performed in the context of repairing and compares our contribution to this work.
- In Chapter 3, we introduce the *DL-Lite* family of ontology languages and we motivate the use of *DL-Lite_A* in the context of our work. We also introduce the Linked Data technologies that are used in our framework.
- In Chapter 4, we describe our framework and algorithms, as well as explain how we address the problems of detecting and resolving invalidities.
- In Chapter 5, we provide details on the implementation of our framework and the applications we have developed, that make use of it.
- In Chapter 6, we describe our experimental evaluation and report on the main conclusions drawn.
- Finally, in Chapter 7, we conclude and describe the future directions of our work.

Chapter 2

Related work

The problem of inconsistencies appearing in KBs can be tackled either by providing the ability to query inconsistent data and get consistent answers (*Consistent Query Answering - CQA*) [10], or by actually *repairing* the KB, which leads to a consistent version of it [11]. Both these approaches have attracted researchers' attention, mostly in the context of relational databases and, lately, in the context of linked data and ontology languages as well.

Computing a repair of an inconsistent KB is, essentially, the problem of performing a minimal set of actions (insertions, deletions, updates) over the KB, in order to render it valid with respect to a given set of integrity constraints. There have been several approaches that deal with repairing inconsistent KBs, with differences in the set of integrity constraints they consider or in the set of actions they allow (e.g., only deletions).

Consistent Query Answering deals with providing the ability to query an inconsistent KB and get as a result only consistent answers. This approach does not materialize a repair of the inconsistent KB, but takes into account, at query execution time, all the possible repairs. There have been many proposed approaches for CQA, considering different semantics, different types of integrity constraints, different ways of computing the repairs.

In the following sections, we will refer to the most notable related works, that have been proposed in the context of relational databases or KBs and deal with the problems of CQA or repairing.

2.1 Repairing and CQA in relational databases

Regarding the repairing of inconsistent relational databases, different semantics have been studied, considering different kinds of constraints.

Chomicki and Marcinkowski [12], following from [13], study the problem of repairing an inconsistent KB, by allowing only tuple deletions (which is also the only action considered in our work) and, in this way, resolving violations of denial constraints and inclusion dependencies, which is a more expressive set of constraints

than the one we consider in this work. However, as it is proven in [12], the unrestricted combination of those constraints leads to intractability issues. In [12], the authors also make use of a *conflict-hypergraph*, which is similar to the approach we follow in our work with the interdependency graph.

On the other hand, CQA in relational databases has been studied in various works dealing with different classes of conjunctive queries and denial constraints, mainly key constraints (e.g., [14, 15]). These works underline the main advantages of using First-Order query rewriting for the validation of integrity constraints. A similar method is also used by our work, in order to translate the constraints to First-Order queries and execute these queries over the KB.

2.2 Repairing and CQA in linked data

In the context of linked data and the corresponding languages and technologies, there has been research on the topic of using ontology languages to encode integrity constraints (ICs) that must be checked over dataset instances.

In [4], the authors present a way to integrate ICs in OWL and they show that IC validation can be reduced to query answering, for integrity constraints that fall into the *SR \mathcal{O} I* DL fragment. This work considers the set of integrity constraints as a different set from the other axioms, with the first being interpreted making the Closed-World Assumption and the latter by making the Open-World Assumption.

A similar approach has been followed in [3]. In [16], the presented approach integrates constraints that come from the relational world (primary-key, foreign-key) into RDF and provides a way to validate these constraints.

IC validation is also an important part of some of the current OWL reasoners, such as Hermit¹, Pellet², FaCT++³, ELK⁴ and others. One of those reasoners, the one that is part of Stardog⁵, provides the ability to express and validate integrity constraints expressed in SPARQL, OWL or SWRL and combines this ability with OWL 2 Reasoning.

However, the above approaches address, essentially, only the KB satisfiability problem and they do not consider detection and repairing of invalidities, which is considered by our work.

During recent years, there has also been some research on CQA for inconsistent knowledge bases expressed in *Description Logics (DL)* languages, using query rewriting techniques. To the best of our knowledge, little work has been performed on the problem of automatically computing a repair for inconsistent KBs.

Regarding CQA on *DLs*, [17] deals with different variants of inconsistency-tolerant semantics to reach a good compromise between expressive power of the semantics and computational complexity of inconsistency-tolerant query answering.

¹<http://hermit-reasoner.com>

²<http://clarkparsia.com/pellet>

³<http://owl.cs.manchester.ac.uk/tools/fact>

⁴<http://www.cs.ox.ac.uk/isg/tools/ELK>

⁵<http://stardog.com/>

More precisely, this work comes to the conclusion that there are inconsistency-tolerant semantics that are FOL-rewritable, and therefore give good complexity behaviour with respect to data complexity.

In the field of diagnosis for *DL-Lite* KBs, there has been some work regarding inconsistency checking. The *DL-Lite_A* reasoner QUONTO [18] has the ability to check the satisfiability of a *DL-Lite_A* KB. However, it does not have the ability to detect the invalid data assertions in the ABox, neither to repair them.

Finally, [19] is the only related work addressing the automatic repairing of an inconsistent *DL-Lite_A* KB, based on the inconsistency-tolerant semantics studied in [17]. The proposed repairing, essentially resolves each invalidity by removing both data assertions that take part in it. On the contrary, our repairing algorithm considers the removal of only one of two interdependent data assertions. Thus, [19] removes more information than necessary from the original KB and, therefore, the proposed repairs are subsumed by the repairs produced by our algorithm. In Chapter 6, we also compare the performance of our framework to the approach in [19], which is evaluated in [1].

Chapter 3

Preliminaries

3.1 Description Logics - *DL-Lite_A*

3.1.1 The Description Logics family of languages

Description Logics (DLs) [20] is a family of knowledge representation formalisms developed over the past four decades and, in recent years, widely used in various domains, such as conceptual modeling, information and data integration, ontology-based data access and the Semantic Web.

In DLs, the important notions of the domain are described by concept and role expressions, which are built from atomic concepts (unary predicates) and atomic roles (binary predicates), using the concept and role constructors provided by the particular DL. Properties of concepts and roles can be specified through inclusion assertions, stating that every instance of a concept (resp., role) is also an instance of another concept (resp., role).

Within a knowledge base (KB), one can see a clear distinction between *intensional knowledge* (or general knowledge about the problem domain) and *extensional knowledge*, which is specific to a particular problem. In a similar manner, a DL KB is typically comprised by two components, a *TBox* and an *ABox*. The TBox contains the intensional knowledge of a KB, in the form of a terminology, and it describes general properties of concepts. On the other hand, the ABox contains the extensional knowledge of a KB, and it describes the knowledge that is specific to the individuals of the domain of discourse. A TBox typically consists of a set of axioms stating the inclusion between pairs of concepts or roles. In an ABox, one can assert membership of objects (i.e., constants) in concepts, or that a pair of objects is connected by a role.

The standard reasoning services over a DL KB include checking its consistency (or satisfiability), instance checking (whether a certain individual is an instance of a concept), and logic entailment (whether a certain constraint is logically implied by the KB).

3.1.2 *DL-Lite* family of DL languages

One of the most important research topics in Description Logics is finding a good trade-off between expressive power and computational complexity of sound and complete reasoning. A significant step in this direction was the introduction of the *DL-Lite* family of ontology languages [21], which is specifically tailored to capture basic ontology languages, conceptual data models (e.g., Entity-Relationship) and object-oriented formalisms (e.g., basic UML class diagrams), while keeping low complexity of reasoning (polynomial in the size of the instances in the ABox).

An important feature of the *DL-Lite* family of languages, is that it is perfectly suited to representing ABox assertions as relations managed in secondary storage by a database management system (DBMS). Thus, the query-answering in *DL-Lite* KBs can be performed by expanding the original query into a set of queries that can be directly evaluated by an SQL (or SPARQL in our case) query engine over the ABox.

The core language of the *DL-Lite* family and its least expressive member is *DL-Lite_{core}*. Concepts and roles are formed using the following syntax:

$$\begin{array}{ll} B \longrightarrow A \mid \exists R & R \longrightarrow P \mid R^- \\ C \longrightarrow B \mid \neg B & E \longrightarrow R \mid \neg R \end{array}$$

where A denotes an *atomic concept*, P an *atomic role*, and P^- the *inverse* of the atomic role P . B denotes a *basic concept*, that is, a concept that can be either an atomic concept or a concept of the form $\exists R$, where R denotes a *basic role*, that is, a role that is either an atomic role or the inverse of an atomic role. Note that $\exists R$ is the standard DL construct of unqualified existential quantification on basic roles. Finally, C denotes a *general concept*, which can be a basic concept or its negation, whereas E denotes a *general role*, which can be a basic role or its negation.

A DL KB $\mathcal{K} = \langle \mathcal{T}, \mathcal{A} \rangle$ represents the domain of interest in terms of two components, a TBox \mathcal{T} and an ABox \mathcal{A} .

A TBox in *DL-Lite_{core}* is formed by a finite set of *inclusion assertions* of the form:

$$B \sqsubseteq C$$

That is, general concepts are allowed to occur on the right-hand side of inclusion assertions, whereas only basic concepts may occur on the left-hand side of inclusion assertions.

An ABox in *DL-Lite_{core}* is formed by a finite set of *membership assertions* on atomic concepts or atomic roles, of the form

$$A(a) \quad P(a, b)$$

stating, respectively, that the object denoted by the constant a is an instance of A and that the pair of objects denoted by the pair of constants (a, b) is an instance of the role P .

3.1.3 DL-Lite_R, DL-Lite_F and DL-Lite_{FR} languages

Before introducing the DL-Lite_A language which is used in the context of our diagnosis and repair framework, we will mention two extensions of DL-Lite_{core}, namely DL-Lite_R and DL-Lite_F, as well as their combination DL-Lite_{FR}, which forms the basis of DL-Lite_A.

DL-Lite_R extends DL-Lite_{core} with the ability of specifying inclusion assertions between roles, of the form

$$R \sqsubseteq E$$

where R and E are defined as above.

DL-Lite_F extends DL-Lite_{core}, by adding the ability of specifying functionality on roles or their inverses. Functionality assertions, used for this purpose, are of the form

$$(\text{funct } R)$$

where R is defined as above.

DL-Lite_{FR} combines the main features of the two DLs described above and extends them. It is structured using the following specification:

- A denotes an *atomic concept*, B a *basic concept*, C a *general concept*, and \top_C denotes the *universal concept*.
- D denotes an *atomic value-domain*, E denotes a *basic value-domain*, F a *general value-domain*, and \top_D the *universal value-domain*.
- P denotes an *atomic role*, Q a *basic role*, and R a *general role*.
- U_C denotes an *atomic concept attribute*, and V_C a *general concept attribute*.
- U_R denotes an *atomic role attribute*, and V_C a *general role attribute*.

Given a concept attribute U_C (resp. a role attribute U_R), we call the *domain* of U_C (resp. U_R), denoted by $\delta(U_C)$ (resp. $\delta(U_R)$), the set of objects (resp. of pairs of objects) that U_C (resp. U_R) relates to values, and we call *range* of U_C (resp. U_R), denoted by $\rho(U_C)$ (resp. $\rho(U_R)$), the set of values that U_C (resp. U_R) relates to objects (resp. pairs of objects). Furthermore, we denote with $\delta_F(U_C)$ (resp. $\delta_F(U_R)$) the set of objects (resp. of pairs of objects) that U_C (resp. U_R) relates to values in the value-domain F .

DL-Lite_{FR} expressions are defined as follows:

1. Concept expressions:

$$\begin{aligned} B &\rightarrow A \mid \exists Q \mid \delta(U_C) \\ C &\rightarrow \top_C \mid B \mid \neg B \mid \exists Q.C \mid \delta_F(U_C) \mid \exists \delta_F(U_R) \mid \exists \delta_F(U_R)^- \end{aligned}$$

2. Value-domain expressions:

$$\begin{aligned} E &\rightarrow D \mid \rho(U_C) \mid \rho(U_R) \\ F &\rightarrow \top_D \mid E \mid \neg E \mid \text{rdf } \textit{DataType} \end{aligned}$$

3. Role expressions:

$$\begin{aligned} Q &\rightarrow P \mid P^- \mid \delta(U_R) \mid \delta(U_R)^- \\ R &\rightarrow Q \mid \neg Q \mid \delta_F(U_R) \mid \delta_F(U_R)^- \end{aligned}$$

4. Attribute expressions:

$$\begin{aligned} V_C &\rightarrow U_C \mid \neg U_C \\ V_R &\rightarrow U_R \mid \neg U_R \end{aligned}$$

A *DL-Lite_{FR}* TBox is formed by a finite of assertions of the following form:

$$\begin{aligned} B &\sqsubseteq C && (\textit{concept inclusion assertion}) \\ Q &\sqsubseteq R && (\textit{role inclusion assertion}) \\ E &\sqsubseteq F && (\textit{value-domain inclusion assertion}) \\ U_C &\sqsubseteq V_C && (\textit{concept attribute inclusion assertion}) \\ U_R &\sqsubseteq V_R && (\textit{role attribute inclusion assertion}) \\ \\ (\textit{funct } P) &&& (\textit{role functionality assertion}) \\ (\textit{funct } P^-) &&& (\textit{inverse role functionality assertion}) \\ (\textit{funct } U_C) &&& (\textit{concept attribute functionality assertion}) \\ (\textit{funct } U_R) &&& (\textit{role attribute functionality assertion}) \end{aligned}$$

A *DL-Lite_{FR}* ABox is formed by a finite of assertions of the following form:

$$A(a), \quad D(c), \quad P(a, b), \quad U_C(a, c), \quad U_R(a, b, c) \quad \textit{membership assertions}$$

3.1.4 *DL-Lite_A* language

Following from the previously described DLs, a *DL-Lite_A* KB is a pair $\langle \mathcal{T}, \mathcal{A} \rangle$, where \mathcal{A} is a *DL-Lite_{FR}* ABox and \mathcal{T} is a *DL-Lite_{FR}* TBox, satisfying the following rules:

1. for every atomic or inverse of an atomic role Q appearing in a concept of the form $\exists Q.C$, the assertions (funct Q) and (funct Q^-) are not in \mathcal{T} ;
2. for every role inclusion assertion $Q \sqsubseteq R$ in \mathcal{T} , where R is an atomic role or the inverse of an atomic role, the assertions (funct R) and (funct R^-) are not in \mathcal{T} ;
3. for every concept attribute inclusion assertion $U_C \sqsubseteq V_C$ in \mathcal{T} , where V_C is an atomic concept attribute, the assertion (funct V_C) is not in \mathcal{T} ;

4. for every role attribute inclusion assertion $U_R \sqsubseteq V_R$ in \mathcal{T} , where V_R is an atomic role attribute, the assertion (funct V_R) is not in \mathcal{T} ;

In the rest of this thesis, we will consider the fraction of $DL-Lite_{\mathcal{A}}$ TBox expressions which can be expressed in OWL syntax, as well as the ABox assertions that can be expressed in the form of RDF triples. Thus, in the following sections, concept and role expressions will be considered to be of the following form:

$$C \longrightarrow A \mid \exists R \quad R \longrightarrow P \mid P^{-}$$

Moreover, a $DL-Lite_{\mathcal{A}}$ TBox will be considered to consist of axioms of the following form:

$$C_1 \sqsubseteq C_2 \quad C_1 \sqsubseteq \neg C_2 \quad R_1 \sqsubseteq R_2 \quad R_1 \sqsubseteq \neg R_2 \quad (\text{funct } R)$$

whereas, a $DL-Lite_{\mathcal{A}}$ ABox will be considered to be a finite set of assertions of the following form:

$$A(a) \quad P(a, b)$$

We make the above simplification, without loss of generality, in order to make the following sections easier to read. However, we should note here, that our approach can be seamlessly extended to the full $DL-Lite_{\mathcal{A}}$ specification.

3.2 FOL-Reducibility

In this section we refer to the notion of FOL-reducibility for KB satisfiability check and query answering. This notion is introduced in [21].

Given an ABox \mathcal{A} , we denote by $db(\mathcal{A}) = (\Delta^{db(\mathcal{A})}, db(\mathcal{A}))$ the *interpretation* defined as follows:

- $\Delta^{db(\mathcal{A})}$ is the nonempty set consisting of all constants occurring in \mathcal{A} ;
- $a^{db(\mathcal{A})} = a$, for each constant a ;
- $A^{db(\mathcal{A})} = \{a \mid A(a) \in \mathcal{A}\}$, for each atomic concept A ; and
- $P^{db(\mathcal{A})} = \{(a_1, a_2) \mid P(a_1, a_2) \in \mathcal{A}\}$, for each atomic role P .

Intuitively, FOL-reducibility of satisfiability (resp., query answering) captures the property that we can reduce satisfiability checking (resp., query answering) to evaluating a FOL query over the ABox \mathcal{A} considered as a relational database, that is, over $db(\mathcal{A})$. Definitions for FOL-Reducibility follow.

Definition 1. KB satisfiability in a DL \mathcal{L} is *FOL-reducible* if, for every TBox \mathcal{T} expressed in \mathcal{L} , there exists a Boolean FOL query q , over the alphabet of \mathcal{T} , such that for every nonempty ABox \mathcal{A} , $\langle \mathcal{T}, \mathcal{A} \rangle$ is satisfiable if and only if q evaluates to false in $db(\mathcal{A})$.

Definition 2. Query answering in a DL \mathcal{L} for unions of conjunctive queries is FOL-reducible if, for every union of conjunctive queries q and every TBox \mathcal{T} expressed in \mathcal{L} , there exists a FOL query q_1 , over the alphabet of \mathcal{T} , such that for every nonempty ABox \mathcal{A} and every tuple of constants \vec{a} occurring in \mathcal{A} , $\vec{a} \in \text{ans}(q, \langle \mathcal{T}, \mathcal{A} \rangle)$ if and only if $\vec{a}^{db(\mathcal{A})} \in q_1^{db(\mathcal{A})}$.

As proven in [22], KB satisfiability in $DL-Lite_{\mathcal{A}}$ is FOL-Reducible. This feature means that one can check the satisfiability of a $DL-Lite_{\mathcal{A}}$ KB by posing the appropriate FOL queries over the ABox, stored in a database (or a triple store in our case). This has an immediate impact on the data complexity of the problem. Indeed, since the FOL queries considered in the above definitions depend only on the TBox (and the query), but not on the ABox, and since the evaluation of a FOL query over an ABox is in AC^0 in data complexity [23], FOL-Reducibility of a problem has an immediate consequence that the problem is in AC^0 in data complexity.

The feature of FOL-Reducibility is used by our framework for performing diagnosis of inconsistencies in a $DL-Lite_{\mathcal{A}}$ KB. We illustrate the diagnosis process in Section 4.1.1 and the corresponding algorithm in Section 4.1.2. In the following subsection we describe the queries that should be posed over the ABox, to diagnose the KB for invalidities.

3.3 $DL-Lite_{\mathcal{A}}$ as a language for the diagnosis of constraint violations

We can distinguish three different types of $DL-Lite_{\mathcal{A}}$ TBox axioms, namely *positive inclusions*, *negative inclusions* and *functionality assertions*. In a simplified - but adequate - manner, positive TBox inclusions are of the form $C_1 \sqsubseteq C_2$ (where C_1 and C_2 are concept expressions) or $R_1 \sqsubseteq R_2$ (where R_1 and R_2 are role expressions). Respectively, negative TBox inclusions are of the form $C_1 \sqsubseteq \neg C_2$ or $R_1 \sqsubseteq \neg R_2$. Functionality assertions are of the form (funct R).

This distinction is important due to the fact that $DL-Lite_{\mathcal{A}}$ makes the Open World Assumption (OWA), which is also considered for the ontology languages of the Semantic Web (such as OWL). Due to the OWA, a positive inclusion can never be violated, because knowledge that is not present in the ABox is not considered as false; therefore, the only interesting (from the diagnosis perspective) axioms are the negative inclusions and functionality assertions.

Nonetheless, positive inclusions are still relevant for the diagnosis process, because they may generate inferred information that should be taken into account. As an example, assume that the TBox contains the constraint $A \sqsubseteq \neg C$ and the axiom $B \sqsubseteq C$ (where A, B and C are atomic concepts), and suppose that the ABox contains both $A(x)$ and $B(x)$ for some x . Even though no constraint is explicitly violated, the combination of the ABox contents with the aforementioned TBox would lead to inferring both $C(x)$ and $\neg C(x)$, i.e., an invalidity. Note that the

positive inclusion $B \sqsubseteq C$, albeit not violated itself, plays a critical role in creating this invalidity.

Rather than capturing such invalidities via the obvious method of computing the closure of the ABox, it is more efficient to identify the constraints implied by the explicitly declared constraints and the positive inclusions in the TBox. In the previous example, we could identify that the constraint $A \sqsubseteq \neg B$ is a consequence of the two explicit axioms in the TBox, so the presence of $A(x)$ and $B(x)$ violates this implicit constraint. This process amounts to computing the closure of negative inclusions and functionality assertions of the TBox ($cln(\mathcal{T})$, also see Section 4.1.2) [21], in other words, the set of all the functionality assertions and the explicit and implicit negative inclusions present in the TBox.

Furthermore, it has been proven that, in order to check the satisfiability of a $DL-Lite_{\mathcal{A}}$ KB (therefore diagnose the KB for invalidities), one has to take into account only the constraints in $cln(\mathcal{T})$. More precisely, it has been shown that an invalidity in a $DL-Lite_{\mathcal{A}}$ KB exists if and only if an ABox assertion contradicts a functionality assertion or a negative inclusion that is present in $cln(\mathcal{T})$ [22].

We should also note here, that constraints expressed in $DL-Lite_{\mathcal{A}}$ allow the presence of interrelated violations, i.e., violations whose potential resolutions coincide; this implies that there are resolutions which resolve more than one violated constraint instance (e.g., negative inclusion or functionality assertion in $cln(\mathcal{T})$) at the same time.

The following example illustrates the derivation of the closure of negative inclusions and functionality assertions from a TBox, and will be used as a running example in the following sections to illustrate the features of our framework:

Example 1. Consider the following $DL-Lite_{\mathcal{A}}$ KB $\mathcal{K} = \langle \mathcal{T}, \mathcal{A} \rangle$:

$$\begin{aligned} \mathcal{T} &= \{(\text{func } P_1), A_1 \sqsubseteq \neg A_2, A_2 \sqsubseteq \exists P_2\} \\ \mathcal{A} &= \{A_1(x_1), A_2(x_1), P_2(x_1, y_1), P_1(x_3, y_2), P_1(x_3, y_3), P_1(x_3, y_4)\} \end{aligned}$$

The closure of negative inclusions and functionality assertions of \mathcal{T} ($cln(\mathcal{T})$), computed in the way that was presented in [22], is the following:

$$cln(\mathcal{T}) = \{(\text{func } P_1), A_1 \sqsubseteq \neg A_2, \exists P_2 \sqsubseteq \neg A_1\}$$

3.4 Linked Data technologies

3.4.1 Resource Description Framework (RDF)

The Resource Description Framework (RDF)¹ is a language for representing information about resources in the World Wide Web and it is a standard model for data interchange on the Web. RDF extends the linking structure of the Web to use URIs to name the relationships between things, as well as the two ends of the link.

¹<http://www.w3.org/TR/2004/REC-rdf-primer-20040210>

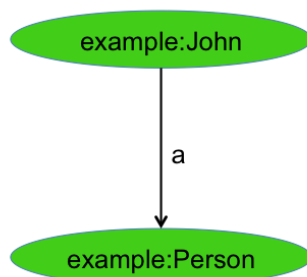


Figure 3.1: The graph view of the triple `<example:John> a <example:Person>`.

This model is referred to as the *triple representation* of an RDF statement, with each statement expressed with a *subject*, a *predicate* and an *object*. The subjects, the predicates and the objects that appear in RDF statements are identified by Uniform Resource Identifiers (URIs).

For example, the triple "`<example:John> a <example:Person>`", states that the resource that exists in the URI `example:John` is a member of the class `example:Person`. The subject of this triple is `example:John`, the predicate is `a` (which states that the subject *IsA* object) and the object is `example:Person`.

The linking structure of an RDF document forms a directed, labeled graph, where the edges represent the named link between two resources, represented by the graph nodes. This *graph view* is the easiest mental model for RDF and it is often used for explaining the structure of an RDF document. The graph view of the triple described above is illustrated in 3.1.

3.4.2 Web Ontology Language (OWL)

Web Ontology Language (OWL)² is a Semantic Web ontology language designed to represent rich and complex knowledge about things, groups of things, and relations between things (in other words, ontologies). OWL is logic-based and it is intended to be used when the information contained in documents needs to be processed by computer applications, as opposed to situations where the content only needs to be presented to humans. OWL is endorsed by the World Wide Web Consortium (W3C) and has attracted academic, medical and commercial interest.

An important feature of OWL is that it can be used, like any other ontology language, to verify the consistency of a dataset with the semantics of the underlying knowledge representation formalism, as well as to infer implicit knowledge from the explicit statements. These tasks are often handled by applications that are called

²<http://www.w3.org/TR/owl2-overview/>

OWL reasoners (e.g. *HermiT*³, *Pellet*⁴, *Stardog*⁵, etc.).

The current version of OWL is *OWL 2*, which has three sublanguages (profiles), namely *OWL 2 EL*, *OWL 2 QL* and *OWL 2 RL*. The OWL 2 QL fragment is based on the *DL-Lite* family of description logics, which was previously discussed. In fact, OWL 2 QL is based on *DL-Lite_R*, giving it nice computational features.

We should make the remark that the techniques and results presented in the following sections have a direct application on OWL 2 QL, i.e., on a standard language of the Semantic Web, that builds on a large user base. Hence, such results are of immediate practical relevance.

3.4.3 SPARQL Protocol and RDF Query Language

SPARQL⁶ is a query language for data stored in the RDF format. As noted before, RDF is a directed, labelled graph data format for representing information in the Web. This specification defines the syntax and semantics of the SPARQL query language. SPARQL can be used to express queries across diverse data sources, whether the data is stored natively as RDF or viewed as RDF via middleware.

SPARQL works as a query language, by matching graph patterns against the data source. The graph pattern may include restrictions and other conditions like optional parts, union, nesting of graphs and filtering of the values of the results. Several SPARQL queries will be presented in the rest of this work.

³<http://hermit-reasoner.com>

⁴<http://clarkparsia.com/pellet>

⁵<http://stardog.com>

⁶<http://www.w3.org/TR/rdf-sparql-query>

Chapter 4

Overview of the diagnosis and repair framework

The purpose of our framework is to automatically perform diagnosis and repairing of invalidities that appear in a $DL-Lite_{\mathcal{A}}$ KB. Our framework consists of two individual components, namely the *diagnosis* and the *repairing* component.

The diagnosis component is responsible for detecting inconsistencies between the ABox and the given TBox, with the integrity constraints it includes. It is responsible, as well, for reporting these invalidities in an intuitive way, in the form of a graph that represents the interdependencies between them.

The repairing component takes as input the graph produced by the diagnosis component and is responsible for automatically computing a repairing delta, as well as for applying this repairing delta to render the KB valid.

The main features of the diagnosis and repair framework are illustrated in Figure 4.1. In the figure, as well as in the definitions presented in this chapter, we use the notation A_i to denote any data assertion appearing in the ABox.

4.1 Diagnosis

4.1.1 Diagnosis component

Before describing the main functionality of the diagnosis component, we must define the meaning of invalidities and invalid data assertions in the context of our framework.

Consider a $DL-Lite_{\mathcal{A}}$ TBox \mathcal{T} , the closure ($cln(\mathcal{T})$) of negative inclusions and functionality assertions of \mathcal{T} and an ABox \mathcal{A} ; then, an *invalidity* is a pair of data assertions in \mathcal{A} which violates a constraint in $cln(\mathcal{T})$. Each of the data assertions that take part in an invalidity, is called an *invalid data assertion*. More formally:

Definition 3. Let $\mathcal{K} = \langle \mathcal{T}, \mathcal{A} \rangle$ be a $DL-Lite_{\mathcal{A}}$ KB and let $cln(\mathcal{T})$ be the closure of negative inclusions and functionality assertions of \mathcal{T} . A pair of data assertions $I = \langle A_1, A_2 \rangle$, where $A_1, A_2 \in \mathcal{A}$, is called an *invalidity* of \mathcal{K} , iff there is some

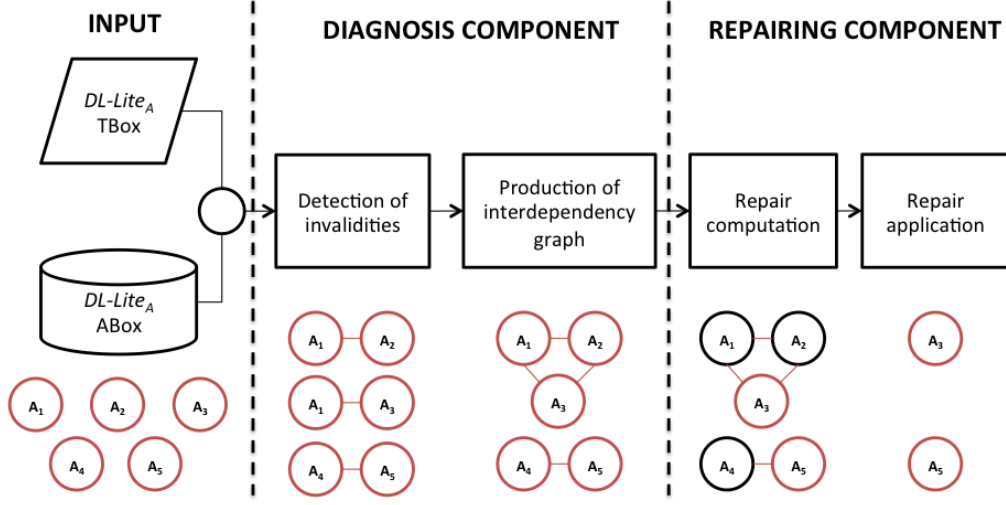


Figure 4.1: The main features of the diagnosis and repair framework.

$c \in \text{cln}(\mathcal{T})$, such that $I \not\equiv c$. In this case, A_1 and A_2 are called *invalid data assertions*.

Following from Definition 3, we can formally define a *consistent* (resp. *inconsistent*) knowledge base:

Definition 4. Let $\mathcal{K} = \langle \mathcal{T}, \mathcal{A} \rangle$ be a $DL\text{-Lite}_{\mathcal{A}}$ KB and let $\text{cln}(\mathcal{T})$ be the closure of negative inclusions and functionality assertions of \mathcal{T} . If \mathcal{K} contains an invalidity I , such that $I \not\equiv c$ for some $c \in \text{cln}(\mathcal{T})$, then \mathcal{K} is called *inconsistent* and we write $\mathcal{A} \not\equiv c$. If \mathcal{K} does not contain any invalidities, it is called *consistent* and we write $\mathcal{A} \equiv \text{cln}(\mathcal{T})$.

The diagnosis component of our framework is responsible for detecting the invalidities in a $DL\text{-Lite}_{\mathcal{A}}$ KB, by posing the constraints in $\text{cln}(\mathcal{T})$ as FOL queries over the ABox. It is also responsible for providing these invalidities as input to the repairing component, in the form of an *interdependency graph*, formally defined below.

Definition 5. Let $\mathcal{K} = \langle \mathcal{T}, \mathcal{A} \rangle$ be a $DL\text{-Lite}_{\mathcal{A}}$ KB, let $\text{cln}(\mathcal{T})$ be the closure of negative inclusions and functionality assertions of \mathcal{T} , let c be a constraint in $\text{cln}(\mathcal{T})$, and let A_1, A_2 be two data assertions in \mathcal{A} . A_1 and A_2 are called *interdependent due to c* , iff $I = \langle A_1, A_2 \rangle \not\equiv c$. The *interdependency graph* of \mathcal{K} is an edge-labeled graph $IG(\mathcal{K}) = (V, E)$, where $V = \{A_i \mid A_i \text{ is an invalid data assertion in } \mathcal{A}\}$ and $E = \{(A_i, A_j, c_k) \mid \langle A_i, A_j \rangle \not\equiv c_k, c_k \in \text{cln}(\mathcal{T})\}$

The use of the interdependency graph as a structure to represent the invalidities that are diagnosed in the KB, gives the ability to get a better grasp of the form

and complexity of interdependencies, to visualize these interdependencies in a way that is easily understood (using graph visualization tools), and to use methods and tools that come from graph theory in order to facilitate the repairing process.

4.1.2 Diagnosis algorithm

The diagnosis algorithm is the core of the diagnosis component and is used to detect all the invalidities in a KB, as well as provide them as output in the form of an interdependency graph. The steps needed to perform diagnosis are illustrated in Algorithm 1.

Algorithm 1 Diagnosis(\mathcal{K})

Input: A *DL-Lite_A* KB $\mathcal{K} = \langle \mathcal{T}, \mathcal{A} \rangle$

Output: The interdependency graph of \mathcal{K} , $IG(\mathcal{K}) = (V, E)$

```

1:  $V, E \leftarrow \emptyset$ 
2: Compute the  $cln(\mathcal{T})$ 
3: for all  $c \in cln(\mathcal{T})$  do
4:    $q_c \leftarrow \delta(c)$ 
5:    $Ans_{q_c} \leftarrow q_c^A$ 
6:   for all  $\langle i_1, i_2 \rangle \in Ans_{q_c}$  do
7:      $V \leftarrow V \cup \{i_1, i_2\}$ 
8:      $E \leftarrow E \cup \{(i_1, i_2, c)\}$ 
9:   end for
10: end for
11: return  $IG(\mathcal{K}) = (V, E)$ 

```

The diagnosis algorithm starts by computing the closure $cln(\mathcal{T})$ of negative inclusions and functionality assertions of the TBox (line 2 of Algorithm 1), in order to get the full set of constraints that need to be checked over the ABox. The process of computing $cln(\mathcal{T})$ is defined below.

Definition 6. Let \mathcal{T} be a *DL-Lite_A* TBox. We call *NI-closure of \mathcal{T}* , denoted by $cln(\mathcal{T})$, the TBox defined inductively as follows:

1. All functionality assertions in \mathcal{T} are also in $cln(\mathcal{T})$.
2. All negative inclusion assertions in \mathcal{T} are also in $cln(\mathcal{T})$.
3. If $C_1 \sqsubseteq C_2$ is in \mathcal{T} and $C_2 \sqsubseteq \neg C_3$ or $C_3 \sqsubseteq \neg C_2$ is in $cln(\mathcal{T})$, then also $C_1 \sqsubseteq \neg C_3$ is in $cln(\mathcal{T})$.
4. If $R_1 \sqsubseteq R_2$ is in \mathcal{T} and $R_2 \sqsubseteq \neg R_3$ or $R_3 \sqsubseteq \neg R_2$ is in $cln(\mathcal{T})$, then also $R_1 \sqsubseteq \neg R_3$ is in $cln(\mathcal{T})$.
5. If $R_1 \sqsubseteq R_2$ is in $cln(\mathcal{T})$ and $\exists R_2 \sqsubseteq \neg C$ or $C \sqsubseteq \neg \exists R_2$ is in $cln(\mathcal{T})$, then also $\exists R_1 \sqsubseteq \neg C$ is in $cln(\mathcal{T})$.

Constraint (c)	Transformation ($\delta(c)$)
$c = A_1 \sqsubseteq \neg A_2$	$\delta(c) = q(x) \leftarrow A_1(x), A_2(x)$
$c = A_1 \sqsubseteq \neg \exists P_1$ (or $c = \exists P_1 \sqsubseteq \neg A_1$)	$\delta(c) = q(x) \leftarrow A_1(x), P_1(x, y)$
$c = A_1 \sqsubseteq \neg \exists P_1^-$ (or $c = \exists P_1^- \sqsubseteq \neg A_1$)	$\delta(c) = q(x) \leftarrow A_1(x), P_1(y, x)$
$c = \exists P_1 \sqsubseteq \neg \exists P_2$	$\delta(c) = q(x) \leftarrow P_1(x, y_1), P_2(x, y_2)$
$c = \exists P_1^- \sqsubseteq \neg \exists P_2^-$	$\delta(c) = q(x) \leftarrow P_1(y_1, x), P_2(y_2, x)$
$c = \exists P_1 \sqsubseteq \neg \exists P_2^-$	$\delta(c) = q(x) \leftarrow P_1(x, y_1), P_2(y_2, x)$
$c = P_1 \sqsubseteq \neg P_2$ (or $c = P_1^- \sqsubseteq \neg P_2^-$)	$\delta(c) = q(x, y) \leftarrow P_1(x, y), P_2(x, y)$
$c = P_1 \sqsubseteq \neg P_2^-$	$\delta(c) = q(x, y) \leftarrow P_1(x, y), P_2(y, x)$
$c = (\text{funct } P)$	$\delta(c) = q(x) \leftarrow P(x, y_1), P(x, y_2)$
$c = (\text{funct } P^-)$	$\delta(c) = q(x) \leftarrow P(y_1, x), P(y_2, x)$

Table 4.1: Transformation of $DL\text{-Lite}_{\mathcal{A}}$ constraints to FOL queries

6. If $R_1 \sqsubseteq R_2$ is in $cln(\mathcal{T})$ and $\exists R_2^- \sqsubseteq \neg C$ or $C \sqsubseteq \neg \exists R_2^-$ is in $cln(\mathcal{T})$, then also $\exists R_1^- \sqsubseteq \neg C$ is in $cln(\mathcal{T})$.
7. If one of the assertions $\exists R \sqsubseteq \neg \exists Q$, $\exists Q^- \sqsubseteq \neg \exists Q^-$, or $Q \sqsubseteq \neg Q$ is in $cln(\mathcal{T})$, then all three such assertions are in $cln(\mathcal{T})$.

Each of the constraints in $cln(\mathcal{T})$ is then transformed to a FOL query using predefined patterns, as defined in Table 4.1 (line 4). The answers to those queries identify the diagnosed invalidities. In Algorithm 1, the execution of each FOL query over the ABox is performed in line 5, where Ans_{q_c} denotes the set of invalidities that break the constraint c . Note that these FOL queries can be easily expressed as SPARQL queries over an ABox stored in a triple store, so that off-the-shelf, optimized tools can be used for query answering (this process is presented in Chapter 5).

The last step of the algorithm encodes the invalidities in the form of an interdependency graph (lines 6-9). This graph is produced by iterating over all invalidities (provided by the previous step of the algorithm) and adding the invalid data assertions as vertices (line 7). For each pair of vertices (representing a pair of interdependent invalid data assertions), an edge connecting them is added and labeled with the constraint that this pair breaks (line 8).

Note that the graph does not contain duplicate vertices, meaning that, an invalid data assertion will appear at most once in the graph, regardless of how many invalidities it is involved in. Also note, however, that for each invalidity that said data assertion is involved in, a different edge connecting its vertex with the vertex that represents the other data assertion of the invalidity, is added to the graph. As a result, we can easily determine how many invalidities each invalid data assertion is involved in. This information is used by the repairing component for the computation of the repairing delta.

The following example illustrates the diagnosis algorithm in action:

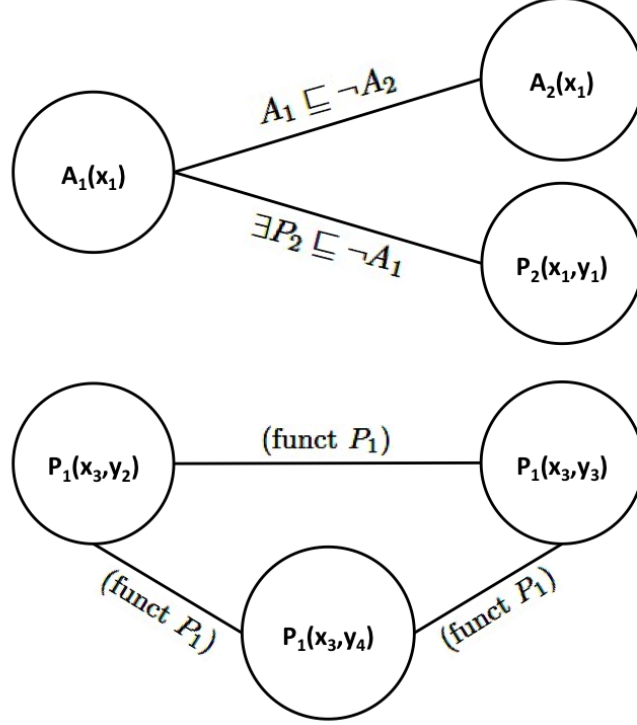


Figure 4.2: Example of an interdependency graph.

Example 2. Consider the KB \mathcal{K} and the $cln(\mathcal{T})$ of Example 1. The corresponding FOL queries to check for invalidities, with respect to the constraints in $cln(\mathcal{T})$, are defined as follows:

$$\begin{aligned} q_1(x) &\leftarrow P_1(x, y) \wedge P_1(x, z) \wedge y \neq z \\ q_2(x) &\leftarrow A_1(x) \wedge A_2(x) \\ q_3(x) &\leftarrow P_2(x, y) \wedge A_1(x) \end{aligned}$$

From the execution of the above three queries over the ABox of Example 1, we get the following sets of invalidities:

$$\begin{aligned} Ans_{q_1} &= \{ \langle P_1(x_3, y_2), P_1(x_3, y_3) \rangle, \\ &\quad \langle P_1(x_3, y_2), P_1(x_3, y_4) \rangle, \\ &\quad \langle P_1(x_3, y_3), P_1(x_3, y_4) \rangle \} \\ Ans_{q_2} &= \{ \langle A_1(x_1), A_2(x_1) \rangle \} \\ Ans_{q_3} &= \{ \langle A_1(x_1), P_2(x_1, y_1) \rangle \} \end{aligned}$$

Figure 4.2 shows the interdependency graph produced by these sets of invalidities.

4.2 Repairing

4.2.1 Repairing component

Given the interdependency graph, the repairing component is responsible for automatically computing and applying a *repairing delta* which leads to a consistent KB. Due to the form of $DL-Lite_{\mathcal{A}}$ constraints, repairing is performed by the deletion of either one of the two invalid data assertions that take part in an invalidity. Thus, in terms of the interdependency graph, resolving an invalidity amounts to removing one of the two vertices that are connected by the edge representing this invalidity. A full repair amounts to repeating this process for all edges in the graph. Formally:

Definition 7. Let $\mathcal{K} = \langle \mathcal{T}, \mathcal{A} \rangle$ be a $DL-Lite_{\mathcal{A}}$ KB, let $cln(\mathcal{T})$ be the closure of negative inclusions and functionality assertions of \mathcal{T} , and let $IG(\mathcal{K}) = (V, E)$ be the interdependency graph of \mathcal{K} . A *repairing delta* of \mathcal{K} is a selection of vertices $V' \subset V$, such that $\mathcal{A} \setminus V' \models cln(\mathcal{T})$. A repairing delta V' is called *minimum*, iff there is no other repairing delta V'' , such that $V'' \subset V'$.

The problem of identifying the minimum repairing delta is actually the well-known problem of finding the minimum *vertex cover* [24]. A vertex cover of a graph is a set of vertices, such that each edge of the graph is incident to at least one vertex of this set. By computing a vertex cover of the interdependency graph, the repairing component computes a repairing delta that, when applied to the dataset, leads to a consistent KB. This is due to the fact that the removal of all the vertices (in other words, all the invalid data assertions) in the vertex cover leads to the removal of all the edges (in other words, all the invalidities) from the graph. Note that we are interested in the minimum, with respect to set inclusion, vertex cover, in order to guarantee minimum impact of the repairing process on the ABox (i.e., remove the minimum amount of data assertions, cf. [7, 8]).

4.2.2 Repairing algorithm

The repairing algorithm takes as input the interdependency graph and is responsible for automatically repairing the KB.

In its first step (line 2 in Algorithm 2), the repairing algorithm breaks the interdependency graph $IG(\mathcal{K})$ into the set of its connected components (denoted as CC in the algorithm). This way, the computation of the vertex cover of $IG(\mathcal{K})$ can be divided into the separate computation of the vertex covers of the connected components, thus it can be parallelized for better performance.

As a next step, the repairing algorithm computes the vertex cover of each of the connected components (lines 3-5). Recall that the computation of the minimum vertex cover is not efficient, as this is a known NP-COMplete problem [25]. However, many approximation algorithms can be used to compute the vertex cover, such as the *2-approximation* algorithm, attributed to Gavril and Yannakakis in [25], or the approximation algorithm presented in [26].

Algorithm 2 Repair($IG(\mathcal{K}), \mathcal{A}$)

Input: An interdependency graph $IG(\mathcal{K})$ and a $DL-Lite_{\mathcal{A}}$ ABox \mathcal{A} **Output:** \mathcal{K} in a consistent state

- 1: $repairing_delta \leftarrow \emptyset$
 - 2: $CC \leftarrow ConnectedComponents(IG(\mathcal{K}))$
 - 3: **for all** $C \in CC$ **do**
 - 4: $repairing_delta \leftarrow repairing_delta \cup GreedyVertexCover(C)$
 - 5: **end for**
 - 6: $\mathcal{A} \leftarrow \mathcal{A} \setminus repairing_delta$
-

We chose to compute the vertex cover in a greedy manner, as presented in [25]. Greedy means that, in each step of the computation, the vertex that is chosen to be included in the cover is the vertex with the highest degree (in other words, the invalid data assertion that is part of the most interdependencies/invalidities). In the case that there exist more than one vertices with the same degree, one of those vertices is arbitrarily chosen. This way, a single vertex cover is returned by the algorithm. This computation is performed in the *GreedyVertexCover* subroutine, which is presented in Algorithm 3.

Algorithm 3 GreedyVertexCover(G)

Input: A graph $G = (V, E)$ **Output:** The vertices in V that belong to an approximate vertex cover of G

- 1: $cover \leftarrow \emptyset$
 - 2: **while** $E \neq \emptyset$ **do**
 - 3: Pick the vertex v with the greatest degree
 - 4: $cover \leftarrow cover \cup v$
 - 5: $V \leftarrow V \setminus v$
 - 6: **end while**
 - 7: **return** $cover$
-

The simple approximation algorithm described above guarantees that the best choice is made locally (i.e., in each step of the algorithm) and is very efficient. It is proven that the above approximation algorithm achieves $O(\log n)$ approximation of the optimal solution, where n is the number of vertices of the graph, with a time complexity of $O(n \log n)$ [25].

The vertices in the computed vertex covers of the connected components represent the invalid data assertions to be removed from the dataset in order to render it valid. It is obvious that, the union of the vertex covers of the connected components of a graph forms a vertex cover of the entire graph. Thus, this union forms the repairing delta (line 4).

The final step of the repairing algorithm is to apply the repairing delta (line 6 of Algorithm 2). This can be performed in a very efficient manner, by posing a single SPARQL-Update query, containing the deletion of all the invalid data assertions

in the repairing delta, over a triple store that stores the ABox (see Chapter 5).

The following concludes the running example for our framework:

Example 3. Consider the interdependency graph of Figure 4.2. The repairing algorithm will compute the following repairing delta:

$$\text{repairing_delta} = \{A_1(x_1), P_1(x_3, y_2), P_1(x_3, y_3)\}$$

After the application of the repairing delta, the ABox \mathcal{A} is in the following state:

$$\mathcal{A} = \{A_2(x_1), P_2(x_1, y_1), P_1(x_3, y_4)\}$$

which forms a consistent KB with respect to the $cln(\mathcal{T})$.

Chapter 5

Diagnosis and repair framework implementation

In this chapter, we provide details on the way we implemented the diagnosis and repair framework, described in the previous chapters, and the tools we used for this purpose. The whole project was developed in the Java programming language¹, using the Eclipse Integrated Development Environment (IDE)².

5.1 Used tools and libraries

5.1.1 OpenLink Virtuoso Open-Source Edition

OpenLink Virtuoso Open-Source Edition³ is a high-performance, cross-platform server for SQL, XML and RDF data management. It includes a virtual database engine, a web services deployment platform, a web application server, as well as SPARQL support over an RDF data store tightly integrated with its relational storage engine.

We chose the Virtuoso Open-Source 7.1.0 server as a triple store for our implementation, as it supports very efficiently the loading of big volumes of data and has a very good query answering performance [27]. It also provides support for the SPARQL/Update (SPARUL) extension of SPARQL, which is used by the repairing component. Moreover, Virtuoso comes with a Jena provider, which supports the interaction between the Jena framework (described below) and the Virtuoso server.

5.1.2 Apache Jena

Apache Jena⁴ (or Jena in short) is a free and open source Java framework for building semantic web and Linked Data applications. The framework is composed

¹<http://www.java.com>

²<http://www.eclipse.org>

³<http://virtuoso.openlinksw.com/dataspace/doc/dav/wiki/Main/>

⁴<http://jena.apache.org>

of different APIs interacting together to process RDF data. Some of the APIs provided by Jena are: the RDF API, the Ontology API, the Inference API, the ARQ API for SPARQL, etc.

In the context of our implementation, we use Jena APIs for three different purposes (more information will be provided in the next separate sections for each component):

- *Ontology API*: We use the Ontology API for loading a $DL-Lite_{\mathcal{A}}$ ontology expressed in OWL syntax, for interacting with that ontology (e.g., finding several different types of assertions) and for exporting the ontology produced after the computation of the closure of negative inclusions and functionality assertions.
- *Inference API*: We use the Inference API for expressing the necessary rules for the computation of the closure of negative inclusions and functionality assertions of the ontology, as well as for applying these rules and inferring the closed ontology.
- *ARQ API*: We use the ARQ API to execute the necessary queries for the diagnosis process and interact with their results, as well as for applying the repairing delta over the triple store that stores the ABox.
- *RDF API*: We use the RDF API for exporting the repairing delta and provide it to the user as an RDF document, for further use.

5.1.3 JUNG - The Java Universal Network/Graph Framework

JUNG⁵ is an open source software library that provides a common and extendible language for the modeling, analysis, and visualization of data that can be represented as a graph or network. It is written in Java, which allows JUNG-based applications to make use of the extensive built-in capabilities of the Java API, as well as those of other existing third-party Java libraries.

The JUNG architecture is designed to support a variety of representations of entities and their relations, such as directed and undirected graphs, multi-modal graphs, graphs with parallel edges, and hypergraphs. It provides a mechanism for annotating graphs, entities and relations with metadata. This facilitates the creation of analytic tools for complex data sets that can examine the relations between entities as well as the metadata attached to each entity and relation.

JUNG also provides a visualization framework that makes it easy to construct tools for the interactive exploration of network data. Users can use one of the layout algorithms provided, or use the framework to create their own custom layouts. In addition, filtering mechanisms are provided which allow users to focus their attention, or their algorithms, on specific portions of the graph.

⁵<http://jung.sourceforge.net>

In the context of our implementation, we use JUNG for the generation and manipulation of the interdependency graph, which is produced by the diagnosis component and is used by the repairing component.

5.1.4 Vaadin framework

Vaadin Framework⁶ is an open source Java web application development framework that helps in creating and maintaining rich web applications. Vaadin supports two programming models: server-side and client-side. Ajax⁷ technology is used at the browser-side to ensure a rich and interactive user experience. On the client-side Vaadin is built on top of and can be extended with Google Web Toolkit⁸.

Vaadin uses Java as the programming language for creating web content. The framework incorporates event-driven programming and widgets, which enables a programming model that is closer to GUI software development than traditional web development with HTML and JavaScript. It uses Google Web Toolkit for rendering the resulting web page.

We use the Vaadin framework to develop the OWLRepair web application, a web application that makes use of our framework implementation. The OWLRepair web application is still under development, but it is in functioning condition. It will be further discussed in 5.3.2.

5.2 Framework architecture

The overall architecture of the framework implementation is illustrated in Figure 5.1. Our implementation consists of three main and independent components: *i) the input component, ii) the diagnosis component and iii) the repairing component.*

In Figure 5.1, each rectangular describes a separate module that handles a specific functionality. Furthermore, the interactions between the modules of each component are illustrated by the arrows that connect them. For example, the output of the module that computes the closure of negative inclusions and functionality assertions of the TBox is provided as input to the module that translates the rules in the closure to their corresponding SPARQL queries, which are then provided as input to the query engine for the query execution. In a similar manner, the module that is responsible for the automatic computation of the repairing delta provides its output as input to the module that applies this repairing delta.

Each component of the overall architecture is implemented as a separate source package and each module is implemented as a Java class. Our implementation follows the Model-View-Controller programming pattern, in order to be more modular.

We have developed two independent applications that use our framework, one that can be run from a console (see Section 5.3.1) and one that can be run as a

⁶<http://vaadin.com>

⁷[http://en.wikipedia.org/wiki/Ajax_\(programming\)](http://en.wikipedia.org/wiki/Ajax_(programming))

⁸<http://www.gwtproject.org>

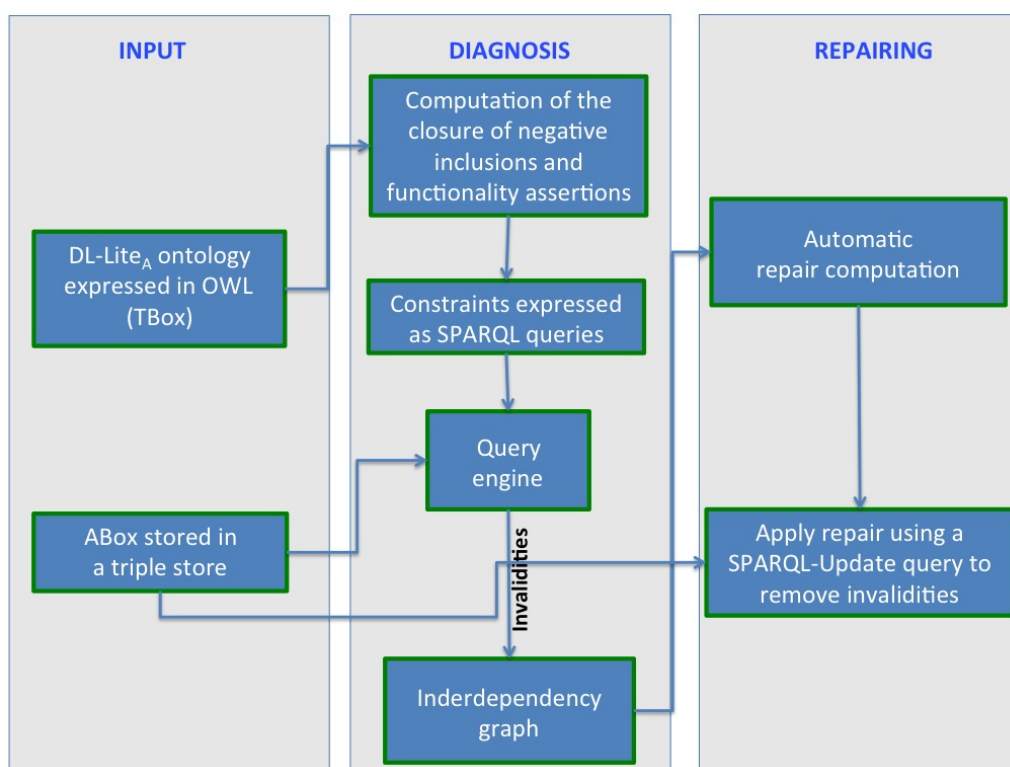


Figure 5.1: The architecture of the diagnosis and repair framework.

web application (see Section 5.3.2). Both of these applications are in functioning condition, but they are under active development.

5.2.1 Input component

The input component of the framework is responsible for loading the ontology that is provided by the user and that will be used as the TBox for the diagnosis process, as well as for connecting to the Virtuoso triple store which stores the ABox that will be queried for invalidities.

Regarding the ontology that is provided by the user, it must be given as an OWL document, using the *DL-Lite_A* specification. The ontology can be given either as a local OWL file or as a URL pointing to an OWL document. In case the ontology contains assertions that are outside the *DL-Lite_A* scope, the framework ignores these assertions and takes into consideration only the ones that correspond to the *DL-Lite_A* TBox assertion rules. After parsing the OWL document, the input component keeps the ontology in a main memory model until the completion of the diagnosis process, using the Ontology API of the Jena framework.

With regard to the ABox, the user is asked to provide the details in order for the input component to establish a connection with the Virtuoso triple store that

stores the data instances. More precisely, the component takes as input: *i*) the IP address of the Virtuoso server deployment, *ii*) the connection port, *iii*) the username and password of the user and, *iv*) optionally, the named graph in which the ABox is stored. After this information is provided, the input component establishes a connection to the triple store which, as it is illustrated in Figure 5.1, is passed to the other two components, as it is used both for diagnosis and for repairing.

5.2.2 Diagnosis component

The implementation of the diagnosis component follows the steps of the diagnosis algorithm, presented in Section 4.1.2. It takes as input (from the input component) the connection to the triple store that stores the ABox and the main memory model that holds the TBox.

Computation of the $cln(\mathcal{T})$

The first step that is performed by the diagnosis component, is the computation of the $cln(\mathcal{T})$. This step is implemented by using the Inference API of the Jena framework and, more precisely, the general purpose rule engine. This general purpose rule-based reasoner can be used to implement both the RDFS and OWL reasoners, but is also available for general use. This reasoner supports rule-based inference over RDF graphs and provides forward chaining, backward chaining and a hybrid execution.

By using the general purpose rule-based reasoner, we express the rules that define the $cln(\mathcal{T})$ and that were described in Section 4.1.2, as generic reasoning rules. Then, the reasoner produces the inferred ontology (in our terms, the $cln(\mathcal{T})$), by applying those rules over the provided TBox, and stores it in the main memory. The following is an example of the syntax of the generic reasoning rules:

Example 4. The following rule, that is part of the definition of $cln(\mathcal{T})$ (see Section 4.1.2):

- If $C_1 \sqsubseteq C_2$ is in \mathcal{T} and $C_2 \sqsubseteq \neg C_3$ or $C_3 \sqsubseteq \neg C_2$ is in $cln(\mathcal{T})$, then also $C_1 \sqsubseteq \neg C_3$ is in $cln(\mathcal{T})$.

is expressed in the context of the Jena general purpose rule-based reasoner, as the following two rules:

- `(?C1 rdfs:subClassOf ?C2), (?C2 owl:disjointWith ?C3) -> (?C1 owl:disjointWith ?C3)`
- `(?C1 rdfs:subClassOf ?C2), (?C3 owl:disjointWith ?C2) -> (?C1 owl:disjointWith ?C3)`

Translation of $cln(\mathcal{T})$ constraints to SPARQL queries - Execution of the SPARQL queries

After the computation of the $cln(\mathcal{T})$, the diagnosis component performs the detection of invalidities in the ABox. This is implemented by executing the FOL queries that correspond to the negative inclusions and functionality assertions in $cln(\mathcal{T})$, as SPARQL queries over the triple store that stores the ABox.

At first, the statements in $cln(\mathcal{T})$ are divided into the following four classes of constraints: *i) concept disjointness*, *ii) property disjointness*, *iii) functional properties*, and *iv) inverse functional properties*. This is performed in the following way:

- If the statement is of the type [C1 owl:disjointWith C2] then it belongs to the *concept disjointness* class of constraints.
- If the statement is of the type [P1 owl:propertyDisjointWith P2] then it belongs to the *property disjointness* class of constraints.
- If the statement is of the type [P1 rdf:type owl:FunctionalProperty] then it belongs to the *functional properties* class of constraints.
- If the statement is of the type [P1 rdf:type owl:InverseFunctionalProperty] then it belongs to the *inverse functional properties* class of constraints.

Then, for each class of constraints, the statements are translated to their corresponding SPARQL query, according to their type.

The procedure of translating the constraints of the concept disjointness class of constraints, is the following: If the statement belongs to the concept disjointness class of constraints (i.e., it is of the type [C1 owl:disjointWith C2]), then the following cases apply, in order to detect the inconsistencies with respect to this statement:

1. Always execute the following SPARQL query:


```
SELECT ?s WHERE {
  ?s rdf:type C1.
  ?s rdf:type C2.
}
```
2. If there is a statement of the type P1 rdfs:domain C1 in the TBox, then execute the following SPARQL query:


```
SELECT ?s ?o WHERE {
  ?s rdf:type C2.
  ?s P1 ?o.
}
```

3. If there is a statement of the type [P1 rdfs:domain C2] in the TBox, then execute the following SPARQL query:

```
SELECT ?s ?o WHERE {  
  ?s rdf:type C1.  
  ?s P1 ?o.  
}
```
4. If there is a statement of the type [P1 rdfs:range C1] in the TBox, then execute the following SPARQL query:

```
SELECT ?s ?o WHERE {  
  ?s rdf:type C2.  
  ?o P1 ?s.  
}
```
5. If there is a statement of the type [P1 rdfs:range C2] in the TBox, then execute the following SPARQL query:

```
SELECT ?s ?o WHERE {  
  ?s rdf:type C2.  
  ?o P1 ?s.  
}
```
6. If there are both statements of the type [P1 rdfs:domain C1] and [P2 rdfs:domain C2] in the TBox, then execute the following SPARQL query:

```
SELECT ?s ?o1 ?o2 WHERE {  
  ?s P1 ?o1.  
  ?s P2 ?o2.  
}
```
7. If there are both statements of the type [P1 rdfs:domain C1] and [P2 rdfs:range C2] in the TBox, then execute the following SPARQL query:

```
SELECT ?s ?o1 ?o2 WHERE {  
  ?s P1 ?o1.  
  ?o2 P2 ?s.  
}
```
8. If there are both statements of the type [P1 rdfs:range C1] and [P2 rdfs:domain C2] in the TBox, then execute the following SPARQL query:

```
SELECT ?s ?o1 ?o2 WHERE {  
  ?o1 P1 ?s.  
  ?s P2 ?o2.  
}
```
9. If there are both statements of the type [P1 rdfs:range C1] and [P2 rdfs:range C2] in the TBox, then execute the following SPARQL query:

```
SELECT ?s ?o1 ?o2 WHERE {  
  ?o1 P1 ?s.
```



```

    ?o2 P2 ?s.
  }

```

The procedure of translating the constraints of the property disjointness class of constraints, is the following: If the statement belongs to the property disjointness class of constraints (i.e., it is of the type [P1 owl:propertyDisjointWith P2]), then the following cases apply, in order to detect the inconsistencies with respect to this statement:

1. Always execute the following SPARQL query:

```

SELECT ?s ?o WHERE {
  ?s P1 ?o.
  ?s P2 ?o.
}

```

2. If there is a statement of the type [P1 owl:inverseOf P3] or [P3 owl:inverseOf P1] in the TBox, then execute the following SPARQL query:

```

SELECT ?s ?o WHERE {
  ?o P3 ?s.
  ?s P2 ?o.
}

```

3. If there is a statement of the type [P2 owl:inverseOf P3] or [P3 owl:inverseOf P2] in the TBox, then execute the following SPARQL query:

```

SELECT ?s ?o WHERE {
  ?s P1 ?o.
  ?o P3 ?s.
}

```

4. If there are both statements of the type [P1 owl:inverseOf P3] and [P2 owl:inverseOf P4] (or their reverse statements) in the TBox, then execute the following SPARQL query:

```

SELECT ?s ?o WHERE {
  ?s P3 ?o.
  ?s P4 ?o.
}

```

The procedure of translating the constraints of the functional properties class of constraints, is the following: If the statement belongs to the functional properties class of constraints (i.e., it is of the type [P1 rdf:type owl:FunctionalProperty]), then the following query is executed:

```

SELECT ?s ?o1 ?o2 WHERE {
  ?s P1 ?o1.
  ?s P1 ?o2.
  FILTER (?o1 != ?o2).
}

```

The procedure of translating the constraints of the inverse functional properties class of constraints, is the following: If the statement belongs to the inverse functional properties class of constraints (i.e., it is of the type [P1 `rdf:type owl:InverseFunctionalProperty`]), then the following query is executed:

```
SELECT ?s1 ?s2 ?o WHERE {
  ?s1 P1 ?o.
  ?s2 P1 ?o.
  FILTER (?s1 != ?s2).
}
```

After the translation has been performed, the SPARQL queries described above are executed against the triple store that stores the ABox, by using the Virtuoso Jena Provider. We should note here that the results of the execution of these queries are variable bindings and not the complete invalid triples. Thus, our implementation wraps these results accordingly, in order to recreate the actual pairs of triples (i.e., invalidities) that appear in the ABox.

Generation of the interdependency graph

With the invalidities in hand, the last step of the diagnosis component is to generate the interdependency graph, on which the repairing component is based. This is implemented by using the JUNG framework.

The interdependency graph is generated by inserting every triple that takes part in an invalidity as a vertex to the graph and by connecting the interdependent triples (see Section 4.1.1 for the definition of interdependency) with an edge. The edge between invalid triples is labeled with the constraint of $cln(\mathcal{T})$ that this invalidity breaks (see Section 4.1.1).

5.2.3 Repairing component

The implementation of the repairing component follows the steps of the repairing algorithm, presented in Section 4.2.2. It takes as input the interdependency graph produced by the diagnosis component and the connection to the triple store that stores the ABox.

Computation of the vertex cover/repairing delta

The first step of the repairing component is the computation of the vertex cover of the interdependency graph, which is, in fact, the repairing delta. The implementation of the computation of the vertex cover follows the steps of the algorithms *Repair* (Algorithm 2) and *GreedyVertexCover* (Algorithm 3), by using methods of the JUNG API.

In order to break the interdependency graph to its connected components, we use the class `WeakComponentClusterer` of the JUNG API. The method `transform`

of this class finds all weak components in a graph, as sets of vertex sets. A weak component is defined as a maximal subgraph in which all pairs of vertices in the subgraph are reachable from one another in the underlying undirected subgraph.

Afterwards, the repairing component computes the vertex cover of each of the connected components of the interdependency graph. These computations are implemented as threads in order to be executed in parallel and improve the performance of the implementation.

The computation is performed by adding to the vertex cover the vertex with the highest degree (i.e., the triple that takes part in the most invalidities), until there are no edges left that are uncovered. This is performed by using an implemented method called `VertexDegreeComparator`, which, in each step of the iteration, compares the vertices of the graph by their degrees.

After the vertex cover of each connected component is computed, all the vertex covers are added up and this set forms the repairing delta.

Application of the repairing delta

The final step of the repairing component (and, as a result, of the diagnosis and repair framework) is the application of the previously computed repairing delta. This is implemented by executing a single SPARQL-Update query, which contains the deletion of every triple in the repairing delta, over the triple store. The way this SPARQL-Update query is formed, is the following:

```
DELETE DATA FROM <named_graph>
{
  TRIPLE_1 .
  TRIPLE_2 .
  ...
  TRIPLE_N .
}
```

After the application of the repairing delta, the knowledge base is consistent and the goal of the diagnosis and repair framework is completed.

5.3 Implemented applications

5.3.1 Console application

The first application that we have implemented on top of the framework that was described in the previous sections is a console application. This application was used for the execution of the experiments that are described in Chapter 6.

The console application comes in the form of a standalone JAR (Java ARchive) file. It can be executed by providing the following VM arguments:

Table 5.2 illustrates the output that is provided by the console application at the end of its execution.

<code>file:</code>	A local file that contains the ontology to be loaded.
<code>inferred:</code>	A local file that contains the (precomputed) $cln(\mathcal{T})$. In this case, the $cln(\mathcal{T})$ is not computed again by the application.
<code>url:</code>	A URL that points to an ontology on the web.
<code>server:</code>	The IP address of the Virtuoso server that hosts the ABox. The default value is <code>localhost</code> .
<code>port:</code>	The connection port of the Virtuoso server. The default value is <code>1111</code> .
<code>graph (Optional):</code>	The named graph that stores the ABox. If it is not provided, all the graphs are used.
<code>user:</code>	The username for the connection to the server. The default value is <code>dba</code> .
<code>pass:</code>	The password for the connection to the server. The default value is <code>dba</code> .

Table 5.1: The input arguments of the console application

<code>Inference time:</code>	The time consumed to compute the $cln(\mathcal{T})$.
<code>Diagnosis time:</code>	The time consumed by the diagnosis component.
<code>Delta computation time:</code>	The time consumed to compute the repairing delta.
<code>Delta application time:</code>	The time consumed to apply the repairing delta.
<code>Number of constraints:</code>	The number of constraints in $cln(\mathcal{T})$.
<code>Executed queries:</code>	The number of queries that were executed by the diagnosis component.
<code>Invalid triples found:</code>	The number of the invalid triples that were diagnosed.
<code>Number of interdependencies:</code>	The number of the edges of the interdependency graph.
<code>Delta size:</code>	The size (in triples) of the repairing delta.

Table 5.2: The output of the console application

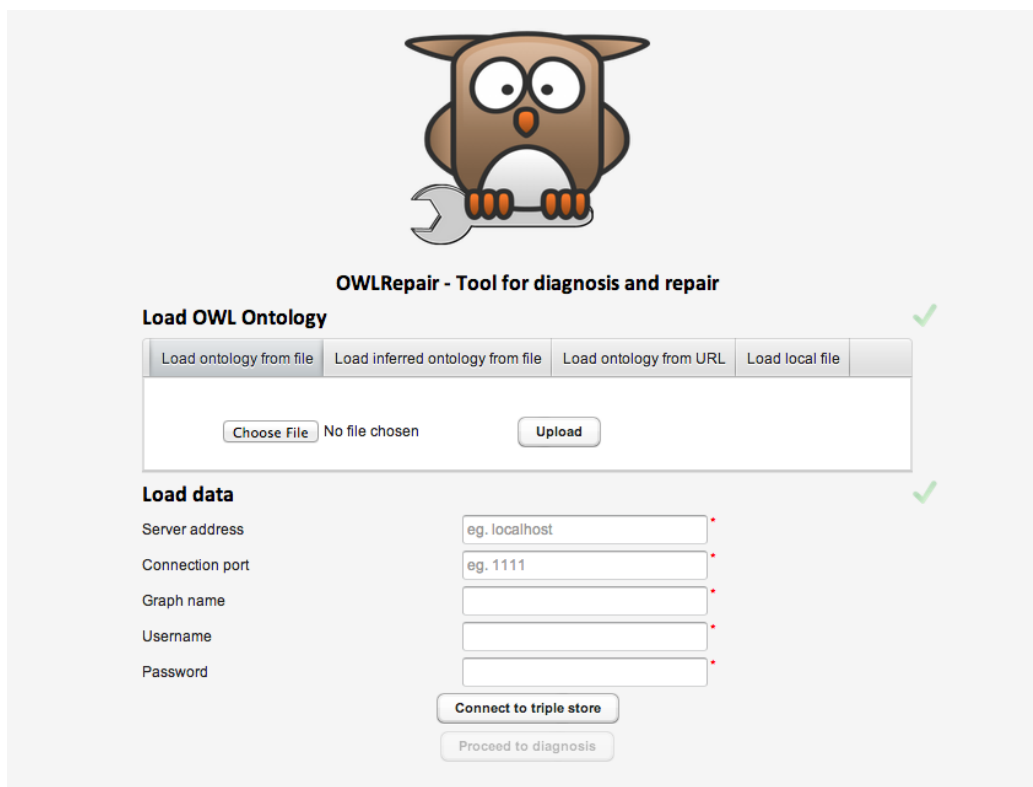
5.3.2 Web application

The other application that we have implemented is a web application, named *OWLRepair*. This application is developed using the Vaadin framework (see Section 5.1.4). *OWLRepair*, essentially, follows the framework implementation that was sketched in the previous section of this chapter. The main feature that is added by *OWLRepair* on top of the implementation is a user-friendly GUI.

Input user interface

The input user interface of *OWLRepair* gives the ability to the user to choose in which way he wants to load the OWL ontology, by providing the following options: *i)upload a file containing the ontology, ii)upload a file containing the $cln(\mathcal{T})$, iii)load the ontology from a URL and, iv)use an ontology that has already been uploaded to the server that hosts OWLRepair*. Moreover, the user is provided with the appropriate fields to fill-in the information about the connection to the triple store. The overall picture of the input user interface of *OWLRepair* is given in Figure 5.2.

A useful feature of the input user interface, is that it provides feedback to the user about the input he has given. This is achieved by giving the user two checkpoints (illustrated as green checkpoints in the figure). One checkpoint is for the loading of the ontology, which is passed (i.e., the visual checkpoint dynamically turns dark green) if the ontology provided by the user is syntactically correct and can be used by the repairing framework. The other checkpoint is for the connection to the triple store, which is passed when the provided information leads to a successful connection. If both checkpoints are passed, the button **Proceed to diagnosis** turns clickable and the user can move on to the diagnosis process.



The screenshot displays the OWLRepair web interface. At the top center is a cartoon owl holding a wrench. Below it, the title reads "OWLRepair - Tool for diagnosis and repair". The interface is divided into two main sections, each with a green checkmark on the right side. The first section, "Load OWL Ontology", contains four tabs: "Load ontology from file", "Load inferred ontology from file", "Load ontology from URL", and "Load local file". Below the tabs is a file selection area with a "Choose File" button, the text "No file chosen", and an "Upload" button. The second section, "Load data", contains five input fields: "Server address" (with "eg. localhost" as a placeholder), "Connection port" (with "eg. 1111" as a placeholder), "Graph name", "Username", and "Password". Below these fields are two buttons: "Connect to triple store" and "Proceed to diagnosis".

Figure 5.2: The input user interface of *OWLRepair*

Diagnosis user interface

The diagnosis user interface of *OWLRepair* is used to provide information to the user about the progress of the diagnosis process, by letting him know how many of the constraints in $cln(\mathcal{T})$ have been diagnosed so far. At the end of the diagnosis process, the user is also given the option to download the $cln(\mathcal{T})$ that has been computed by the diagnosis component. The two different pages of the diagnosis user interface are presented in Figure 5.3 and Figure 5.4.

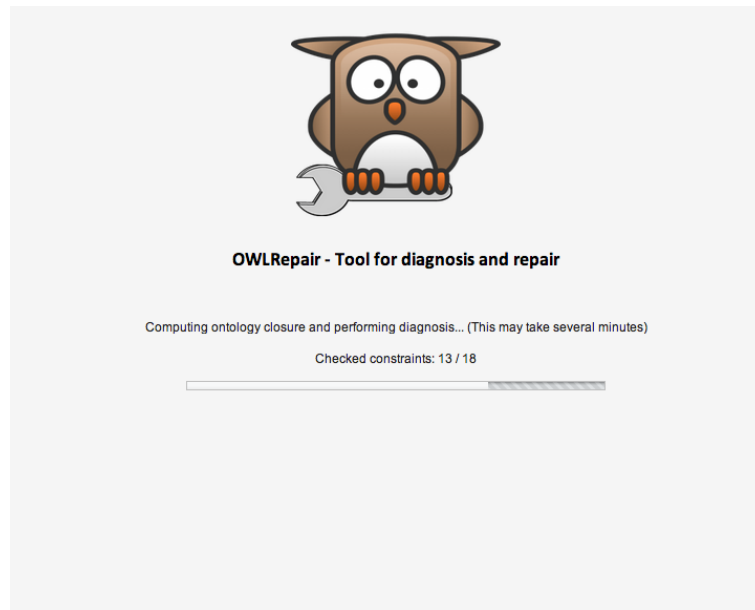


Figure 5.3: The diagnosis user interface of *OWLRepair*

Repair user interface

The repair user interface of *OWLRepair* presents the computed repairing delta in a user-friendly manner, giving the user the choice to make alterations to the triples that will be removed.

The repairing delta is given as a table of triples which can be sorted with respect to the subject, the predicate or the object (see Figure 5.5). The user has the option to click on a triple and see the available alternative option in order to exclude this triple from the repairing delta (i.e., keep it in the triple store). This is implemented by searching the interdependency graph for triples that are interdependent with the selected triple and are not currently included in the repairing delta. Thus, the alternative option would be to include in the repairing delta, all those triples that are interdependent with the selected triple. This interface is illustrated in Figure 5.6.

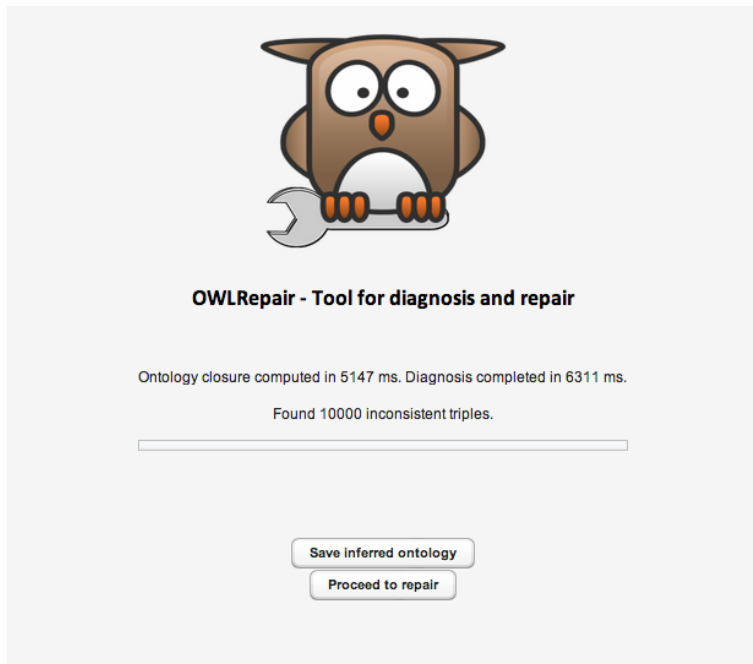


Figure 5.4: The diagnosis user interface of *OWLRepair*

Finally, the user is given the option to save the repairing delta for future reference or use, and to apply the repairing delta on the triple store.

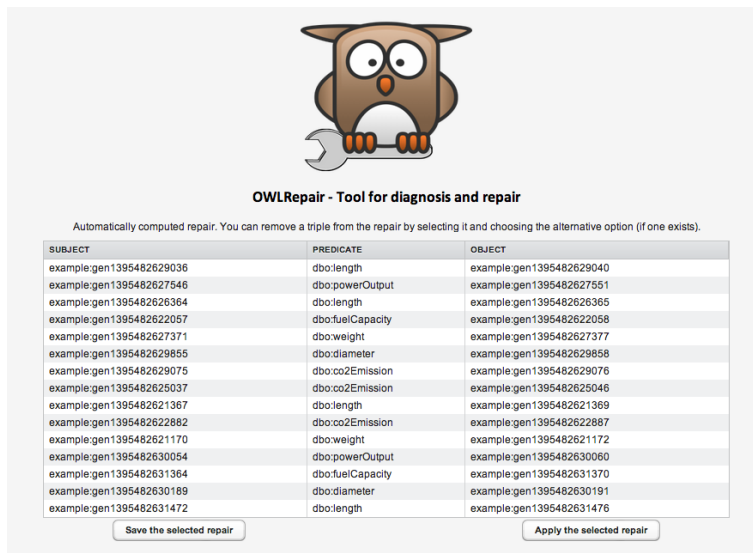
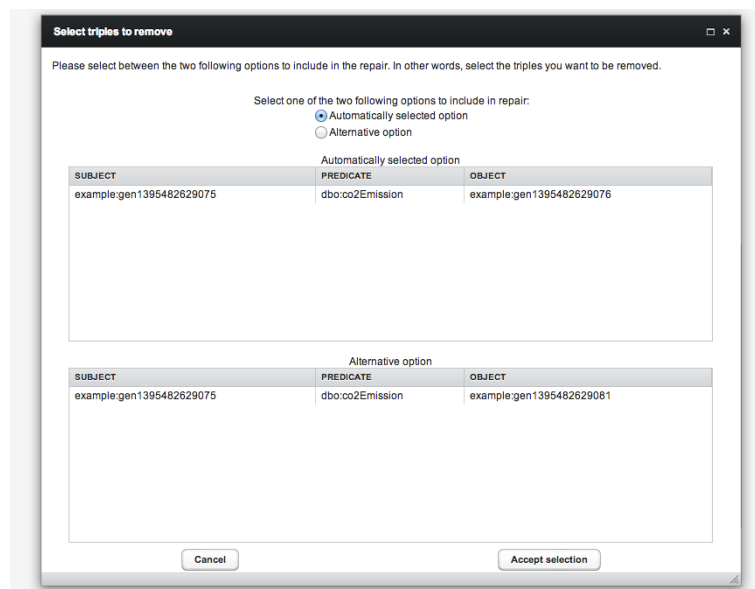


Figure 5.5: The repair user interface of *OWLRepair*

Figure 5.6: The repair user interface of *OWLRepair*

Chapter 6

Evaluation of the framework

6.1 Environment of experiments

The experimental evaluation of our framework was performed using the console application that we have developed (c.f. Chapter 5). The system that we used to perform the experiments had the following specification: AMD Opteron™ 3280 8-core CPU with 24GB RAM (we allocated 8GB for the JVM), running Ubuntu Server 12.04. We used version 07.10 of Virtuoso Open-Source Edition Server. The experiments were performed with the use of 3x¹, a tool for executing exploratory experiments.

We have performed several experiments in order to measure the performance and scalability of our framework, as well as to determine those factors that play a decisive role in the performance of the different phases of the process. We also performed a set of experiments to compare the performance of our framework to the experimental evaluation of the approach in [19] (the evaluation of this work is presented in [1]).

For our experiments, we used different combinations of TBoxes (hereafter, referred to as *ontologies*) and ABoxes (hereafter, referred to as *datasets*). More specifically, we performed four sets of experiments: *i*) the first set of experiments verified that our framework can handle millions of violations in real-world datasets that scale up to more than 2 billion triples, considering hundreds of thousands of constraints, *ii*) the second set quantified the impact of the dataset size on performance, by using real ontologies with constraints and synthetic datasets of varying sizes, *iii*) the third set quantified the impact of the number of invalid data assertions on performance, by using real ontologies with constraints and synthetic datasets with varying number of invalid data assertions and, *iv*) the fourth set of experiments compared the performance of our framework to the performance of the approach in [19].

In all of the above sets of experiments, we measured the time needed to run the diagnosis algorithm and produce the interdependency graph (*diagnosis time*),

¹<http://netj.github.io/3x>

the time needed by the repairing algorithm to compute the repairing delta (*repair computation time*) and the time needed to apply this repairing delta on the dataset, using a SPARQL-Update query (*repair application time*). All of our experiments were run in sets of 5 hot runs and the average times were taken.

6.2 Description of used ontologies

6.2.1 1st-3rd sets of experiments

As ontologies for the first three sets of experiments, we used two versions of the DBpedia ontology, namely versions 3.6 and 3.9. These two ontology versions already contain different amounts and types of constraints, as illustrated in Table 6.1. More specifically, for each ontology version, Table 6.1 shows information on how many functional and (concept/domain/range) disjointness constraints exist in the original ontology, as well as how many of these exist in the closure of negative inclusions of the ontology ($cln(\mathcal{T})$)² and how many queries need to be executed for diagnosis. We should note here that we were unable to find any real ontologies that contained property disjointness constraints, which is the only remaining type of constraints that can be captured by $DL-Lite_{\mathcal{A}}$. For running our experiments, we used the two ontology versions, together with both real and synthetic datasets.

Ontology version	Constraints		Constraints in $cln(\mathcal{T})$		Queries
	Functional	Disjointness	Functional	Disjointness	
DBpedia 3.6	18	0	18	0	18
DBpedia 3.9	26	17	26	323389	323415

Table 6.1: Information on constraints contained in different DBpedia ontology versions.

6.2.2 4th set of experiments

For the fourth set of experiments, we used the Lehigh University Benchmark (LUBM) ontology. We should note here that the original LUBM ontology does not contain any negative inclusions or functionality assertions, so it can not cause any invalidities. For this reason, we modified the LUBM ontology in the same way as it was modified in [1], by adding the following axioms:

²The big difference in the amount of disjointness constraints between the original DBpedia 3.9 ontology and its $cln(\mathcal{T})$ is caused by the many positive inclusions and their interaction with the negative inclusions during the computation of $cln(\mathcal{T})$.

```

:takesCourse owl:propertyDisjointWith :teacherOf
:takesCourse owl:propertyDisjointWith :worksFor
:Student owl:disjointWith :Professor
:Student owl:disjointWith :Organization
:Course owl:disjointWith :Person
:ResearchGroup owl:disjointWith :University
:ResearchGroup owl:disjointWith :Institute
:Person owl:disjointWith :Publication
:ResearchAssistant owl:disjointWith :FullProfessor
:subOrganizationOfInverse owl:inverseOf :subOrganizationOf
:subOrganizationOf owl:propertyDisjointWith :subOrganizationOfInverse

```

6.3 Synthetic data generator for diagnosis and repair

In our effort of evaluating our framework for performance and scalability, we faced the problem of not being able to find a sufficient amount of real-world ontology and dataset combinations to take as input. The combinations that we found either did not include any invalidities in their datasets, or they did not have a sufficient amount of constraints in their ontologies. The only real-world combinations, that contained both constraints and invalidities based on these constraints, were the various DBpedia versions. However, these were also not enough to evaluate the performance of our framework in various different sizes of datasets and numbers of invalidities.

Due to this fact, we developed a basic synthetic data generator that would help us in the process of evaluating our framework. This data generator is described in the rest of this section.

6.3.1 Overview

The synthetic data generator takes as input a TBox that contains negative inclusions and/or functionality assertions and, based on the constraints in $cln(\mathcal{T})$ and the relations between those constraints, produces a uniformly distributed set of invalidities.

The two parameters of the generator, besides the TBox, are the number of invalid triples and the total size of the synthetic dataset (valid and invalid triples) to be produced, as the generator not only produces invalidities, but also has the ability to produce synthetic triples that are valid with respect to the given constraints.

6.3.2 Triple pattern graph

The first task of the synthetic data generator is to generate a graph which contains all the pairs of triple patterns that can lead to an inconsistency.

After the computation of the $cln(\mathcal{T})$, the data generator iterates through all of the constraints. For each constraint, it generates a pair of triple patterns, which, if instantiated appropriately in the dataset, would break the constraint. The triple patterns are formed by using the appropriate URI as predicate and the appropriate URIs or placeholders as subject and object.

The two triple patterns are inserted in a graph called *triple pattern graph* and are connected with each other with an edge that describes in which cases the simultaneous presence of their instances in the dataset would break the constraint. The edge describes which parts (subject or object) of the two triple patterns should be the same in their instantiations, in order to lead to an inconsistency. We should note that one triple pattern can be connected to itself (functional constraints).

An illustrative example of the above process is provided below:

Example 5. Suppose that the $cln(\mathcal{T})$ is the same as in the Example 1:

$$cln(\mathcal{T}) = \{(\text{func } P_1), A_1 \sqsubseteq \neg A_2, \exists P_2 \sqsubseteq \neg A_1\}$$

The pairs of triple patterns generated for each of the above constraints will be the following:

- (func P_1) will generate the following pair of triple patterns:
`%s% P1 %o%`
`%s% P1 %o%`
- $A_1 \sqsubseteq \neg A_2$ will generate the following pair of triple patterns:
`%s% rdf:type A1`
`%s% rdf:type A2`
- $\exists P_2 \sqsubseteq \neg A_1$ will generate the following pair of triple patterns:
`%s% P2 %o%`
`%s% rdf:type A1`

The triple pattern graph generated by the above constraints is illustrated in Figure 6.1.

Algorithm for the triple pattern graph generation

The algorithm for the generation of the triple pattern graph, given a $DL\text{-}Lite_{\mathcal{A}}$ $cln(\mathcal{T})$, is presented in Algorithm 4. In this algorithm, for each constraint in $cln(\mathcal{T})$, the corresponding triple patterns are created and are inserted in the graph. For each pair of triple patterns created, an edge connecting their two vertices is also added to the graph, describing the similarity that the instantiations of the two patterns should have, in order to cause an invalidity (e.g., an edge with label `SubjectWithSubject` means that, in order to cause an invalidity, the instantiations of the two patterns that are at the two ends of this edge should have the same URI as subject). The translation of $cln(\mathcal{T})$ constraints to vertices and edges of the triple pattern graph is described in Table 6.2.

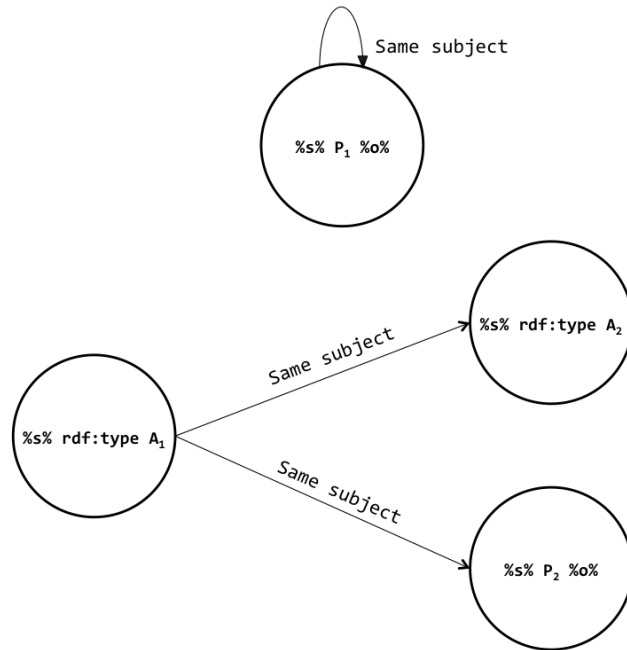


Figure 6.1: Example of a triple pattern graph.

6.3.3 Generation of invalidities

After the generation of the triple pattern graph, the data generator proceeds to the generation of invalidities, based on it. As a first step, the generator picks a random vertex of the triple pattern graph. For this vertex, an instance of the triple pattern is created (i.e., new, unique resource URIs that replace the subject and object placeholders are generated and a new triple is produced). Afterwards, a random distance from the chosen vertex is picked, with values between one and four. Up to this distance, the generator visits (recursively) a random amount of neighbors and instantiates their triple patterns, by giving the same URI in their connected resources (subject or object) and new unique URIs for the resources that are left³.

After traveling the randomly picked distance and visiting the randomly picked neighbors, another vertex from the triple pattern graph is randomly chosen and the process described above is restarted, until the number of produced invalid triples reaches the number that was given as input.

Algorithm for the generation of invalidities

The process previously described is illustrated in Algorithms 5 and 6. In Algorithm 5, the first step is to pick a random vertex from the triple pattern graph

³We use the maximum distance of 4 in order to simulate real datasets, which, in general, do not have larger paths in their interdependency graphs

Constraint c	$V1(c)$	$V2(c)$	$E(c)$
(funct P)	%s% P %o%	%s% P %o%	$V1(c), V2(c), \text{SubjectWithSubject}$
(funct P^-)	%s% P %o%	%s% P %o%	$V1(c), V2(c), \text{ObjectWithObject}$
$C_1 \sqsubseteq \neg C_2$	%s% rdf:type %C1%	%s% rdf:type %C2%	$V1(c), V2(c), \text{SubjectWithSubject}$
$C_1 \sqsubseteq \neg \exists P$	%s% rdf:type %C1%	%s% P %o%	$V1(c), V2(c), \text{SubjectWithSubject}$
$C_1 \sqsubseteq \neg \exists P^-$	%s% rdf:type %C1%	%s% P %o%	$V1(c), V2(c), \text{SubjectWithObject}$
$\exists P \sqsubseteq \neg C_1$	%s% P %o%	%s% rdf:type %C1%	$V1(c), V2(c), \text{SubjectWithSubject}$
$\exists P^- \sqsubseteq \neg C_1$	%s% P %o%	%s% rdf:type %C1%	$V1(c), V2(c), \text{ObjectWithSubject}$
$\exists P_1 \sqsubseteq \neg \exists P_2$	%s% P ₁ %o%	%s% P ₂ %o%	$V1(c), V2(c), \text{SubjectWithSubject}$
$\exists P_1 \sqsubseteq \neg \exists P_2^-$	%s% P ₁ %o%	%s% P ₂ %o%	$V1(c), V2(c), \text{SubjectWithObject}$
$\exists P_1^- \sqsubseteq \neg \exists P_2$	%s% P ₁ %o%	%s% P ₂ %o%	$V1(c), V2(c), \text{ObjectWithSubject}$
$\exists P_1^- \sqsubseteq \neg \exists P_2^-$	%s% P ₁ %o%	%s% P ₂ %o%	$V1(c), V2(c), \text{ObjectWithObject}$
$P_1 \sqsubseteq \neg P_2$	%s% P ₁ %o%	%s% P ₂ %o%	$V1(c), V2(c), (\text{SubjectWithSubject}, \text{ObjectWithObject})$
$P_1^- \sqsubseteq \neg P_2^-$	%s% P ₁ %o%	%s% P ₂ %o%	$V1(c), V2(c), (\text{SubjectWithSubject}, \text{ObjectWithObject})$
$P_1 \sqsubseteq \neg P_2^-$	%s% P ₁ %o%	%s% P ₂ %o%	$V1(c), V2(c), (\text{SubjectWithObject}, \text{ObjectWithSubject})$
$P_1^- \sqsubseteq \neg P_2$	%s% P ₁ %o%	%s% P ₂ %o%	$V1(c), V2(c), (\text{ObjectWithSubject}, \text{SubjectWithObject})$

Table 6.2: Translation of $cln(\mathcal{T})$ constraints to vertices and edges of the triple pattern graph

Algorithm 4 TriplePatternGraph($cln(\mathcal{T})$)

Input: A *DL-Lite_A* $cln(\mathcal{T})$

Output: The triple pattern graph $TPG = (V, E)$ that corresponds to $cln(\mathcal{T})$

- 1: $V, E \leftarrow \emptyset$
 - 2: **for all** $c \in cln(\mathcal{T})$ **do**
 - 3: $V \leftarrow V \cup \{V1(c), V2(c)\}$
 - 4: $E \leftarrow E \cup \{E(c)\}$
 - 5: **end for**
 - 6: **return** $TPG = (V, E)$
-

(line 3) and instantiate its triple pattern (lines 4-13). The *NewResourceURI* routine (lines 9 and 12) generates a unique resource URI to assign to the subject or the object of the instantiated triple pattern. Afterwards, the triple is added to the set of invalid triples (line 14), a random distance between one and four is chosen (line 15) and the recursive routine *RandomWalk* is called (line 16). The arguments of *RandomWalk* are: *i*)the currently chosen vertex of the triple pattern graph, *ii*)the subject that was assigned to the instantiation of the triple pattern, *iii*)the object that was assigned to the instantiation of the triple pattern, *iv*)the set of already produced invalid triples, *v*)the current distance that has been traveled from the initial vertex, *vi*)the distance that was randomly picked as limit and, *vii*)the total number of invalid triples to be produced.

The *RandomWalk* recursive routine is presented in Algorithm 6. At first, the routine checks if the distance already traveled from the initial vertex is smaller than the distance that was previously randomly picked (line 1) and, if true, it picks a random number of neighbors of the vertex that called the routine (line 2).

Afterwards, the routine begins an iteration for as many times as the randomly picked number of neighbors and while the invalid triples that have been produced are less than the invalid triples that have were asked (lines 3 and 4). The first task

Algorithm 5 InvalidationGeneration(TPG, n)

Input: A triple pattern graph $TPG = (V, E)$ and the number of invalid triples to be produced

Output: A set INV of n invalid triples

```

1:  $INV \leftarrow \emptyset$ 
2: while  $|INV| \leq n$  do
3:   Randomly pick a vertex  $v \in V$ 
4:   Create a new triple  $t$ 
5:    $\text{subject}(t) = \text{subject}(v)$ 
6:    $\text{predicate}(t) = \text{predicate}(v)$ 
7:    $\text{object}(t) = \text{object}(v)$ 
8:   if  $\text{subject}(t) = \text{NewResourceURI}$  then
9:      $\text{subject}(t) = \text{NewResourceURI}$ 
10:  end if
11:  if  $\text{object}(t) = \text{NewResourceURI}$  then
12:     $\text{object}(t) = \text{NewResourceURI}$ 
13:  end if
14:   $INV \leftarrow INV \cup t$ 
15:   $dist =$  a random number between 1 and 4
16:   $\text{RandomWalk}(v, \text{subject}(t), \text{object}(t), INV, 1, dist, n)$ 
17: end while
18: return  $INV$ 

```

inside the iteration, is to randomly pick a neighbor vertex of the vertex that called the routine (line 5) and instantiate its triple pattern (lines 6-23). The algorithm performs the necessary checks to determine what kind of edge connects the two vertices (the one that called the routine and the one that was picked from the neighbors) and, according to the type of the vertex, it gives the newly created triple its subject (lines 10-16) and its object (lines 17-23). After the instantiation, the generated triple is added to the set of invalid triples (line 24) and the routine calls itself with new arguments, in order to continue traveling in the graph until the distance limit is reached.

6.4 Description of used datasets

6.4.1 1st-3rd sets of experiments

As real data, we used the two DBpedia datasets that correspond to the two aforementioned ontology versions, namely DBpedia 3.6 and DBpedia 3.9 (multi-language versions), stored in a local Virtuoso instance. The DBpedia 3.6 dataset contains 541M triples, whereas the DBpedia 3.9 dataset contains more than 2 billion triples. The real data were used in the 1st set of experiments.

In order to generate synthetic data, we used the two DBpedia ontology ver-

% of invalidities \ # of Universities	1	5	10	20
1	101560	630842	1285431	2715200
5	105836	657404	1339554	2811400
10	111716	693926	1413974	2986720
20	125681	780667	1590721	3360060

Table 6.3: Sizes (in triples) of the datasets used in the 4th set of experiments

sions and produced new datasets that contained a defined numbers of invalid data assertions, using the synthetic data generator previously described. More specifically, for each of the ontology versions, we generated two different sets of synthetic datasets that contained invalid data assertions. The first set of datasets had a fixed number of invalid data assertions (10K triples) and a varying dataset size (500K-5M triples, with a step of 500K triples). The second set of datasets had a fixed dataset size (10M triples) and a varying number of invalid data assertions (50K-500K triples, with a step of 50K triples). The above two sets of datasets were used in the 2nd and 3rd set of experiments respectively.

6.4.2 4th set of experiments

The data used in the 4th set of experiments were partly produced by the LUBM data generator (UBA) and partly produced by our data generator. We used UBA to generate valid data, with respect to the original LUBM ontology, and we used our data generator to produce invalidities, with respect to the axioms that we added to the original LUBM ontology.

We produced four different versions for every original dataset that was produced by UBA (1 university, 5 universities, 10 universities and 20 universities). The four different versions of each dataset contained different percentages of invalid data assertions (1% of the dataset, 5%, 10% and 20%). The sizes of the datasets we used in this set of experiments are illustrated in Table 6.3. The sizes have insignificant differences with the ones presented in [1], which are caused by the different tools we used to produce the invalidities. However, the numbers of invalid data assertions that appear in our datasets are true to the percentages of the dataset sizes.

6.5 Scalability and performance

1st set of experiments

The first set of experiments was performed to verify the scalability of our framework in real-world settings, with datasets of billions of triples and with large numbers of constraints (up to hundreds of thousands). For this purpose, we used the two

complete versions of DBpedia (both ontology and real dataset, namely versions 3.6 and 3.9). For each of these two versions, we measured the diagnosis time, the repair computation time, the repair application time and the total time. We also measured the number of invalid data assertions that appear in the datasets, to see how well our framework scales with respect to that, as well as the size of the repairing delta, to see whether a manual repair by the curator would be feasible in this context.

The results of this set of experiments are illustrated in Table 6.4. In the table, *IDA* denotes the number of invalid data assertions, t_d denotes the diagnosis time, $t_{r.c.}$ the repair computation time, $t_{r.a.}$ the repair application time and t_t the total time needed for diagnosis and repairing. All times are in milliseconds.

Ontology	Dataset (Triples)	IDA	Delta	t_d	$t_{r.c.}$	$t_{r.a.}$	t_t
DBpedia 3.6	DBpedia 3.6 (541M)	1109	749	2440	402	219	3061
DBpedia 3.9	DBpedia 3.9 (>2B)	1020199	717798	9610319	27190191	1415329	38215839

Table 6.4: Experiments performed on real datasets.

The results show that our framework is scalable, for both large datasets and big numbers of invalid data assertions, and that it can be applied in real-world settings, as the total time for diagnosis and repairing goes up to about 11 hours. It also proves the important fact that, already deployed and massively used reference KBs, such as DBpedia, don't have sufficient mechanisms for preventing the introduction of invalid data or for detecting and repairing such invalid data. Moreover, our experiments illustrate that the number of invalid data assertions and the size of the repairing delta would be prohibitive for repairing by manual curation.

2nd set of experiments

As a next set of experiments, we used the two versions of the DBpedia ontology and, for each of them, we used the corresponding set of synthetic datasets of varying dataset sizes and fixed number of invalid data assertions, that we had previously generated. The results of this set of experiments can be found in Figures 6.2-6.3. We should note here that some of the curves in the graphs are difficult to distinguish, either because they are too close to the start of the x-axis (such as the repair computation time and the repair application time in Figure 6.3), or because they are too close with another curve (such as the diagnosis time and the total time in Figure 6.3).

From the results of this set of experiments, we can come to the conclusion that diagnosis time grows linearly with respect to the dataset size and that it is the dominating impact factor of the total time, when the number of invalid data assertions is fixed. This is an important conclusion because it shows that, overall, our framework scales linearly with respect to the dataset size.

Moreover, another conclusion is that the number of constraints has a major impact on the diagnosis time and the total time, which are two orders of magnitude

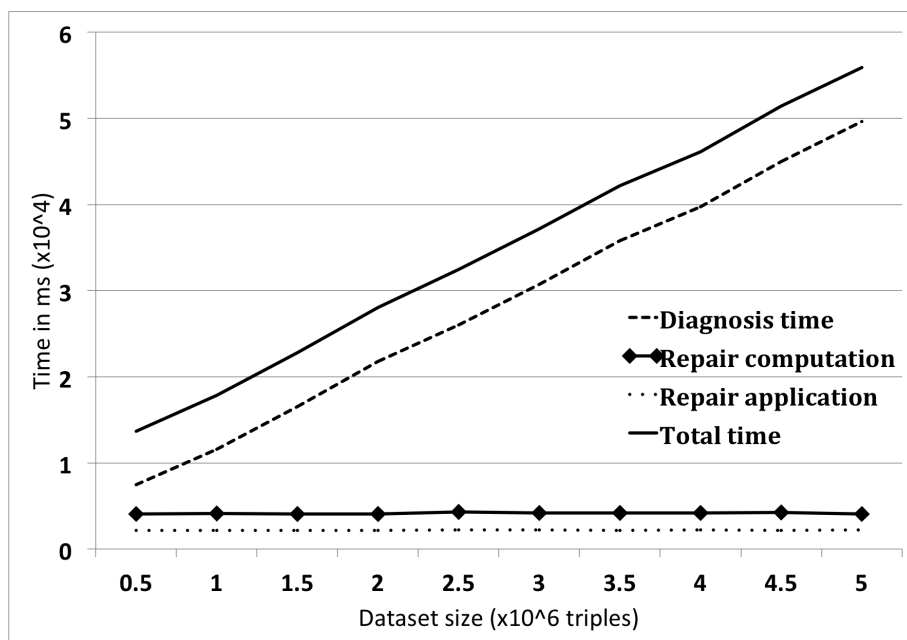


Figure 6.2: Performance for DBpedia ontology version 3.6 with datasets of varying dataset sizes and fixed number of invalid data assertions (10K triples).

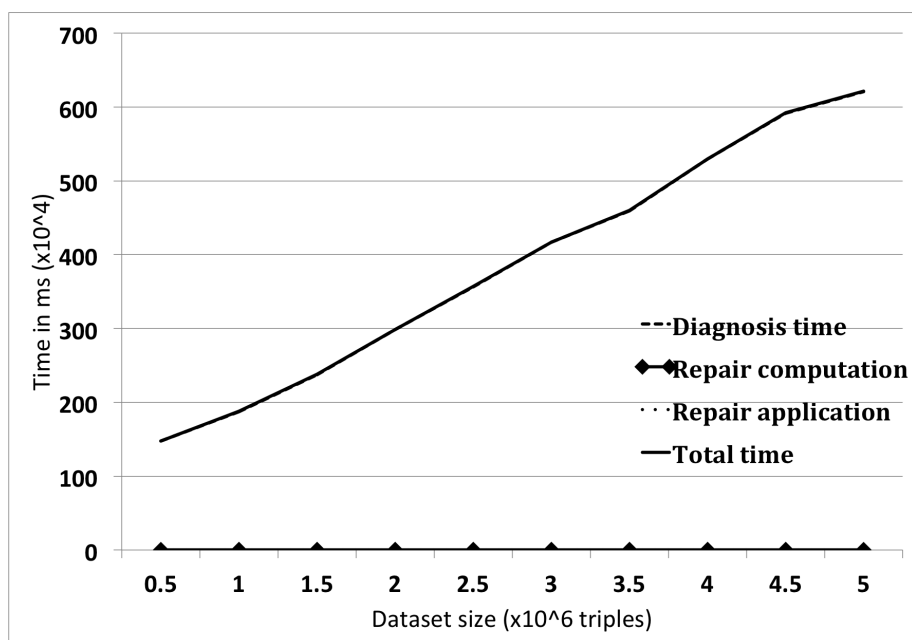


Figure 6.3: Performance for DBpedia ontology version 3.9 with datasets of varying dataset sizes and fixed number of invalid data assertions (10K triples).

larger in the experiments performed with the version 3.9 of the DBpedia ontology, that contains 323415 constraints, than in the experiments performed with the version 3.6, that contains 18 constraints.

3rd set of experiments

As a 3rd set of experiments, we combined the two versions of the DBpedia ontology with the corresponding sets of synthetic datasets, that contained varying number of invalid data assertions and had fixed dataset size. The results of this set of experiments are illustrated in Figures 6.4-6.5.

From these results, we can conclude that the number of invalid data assertions has no immediate impact on the diagnosis time. On the contrary, this number is the main impact factor of the repair computation time. That was an expected behavior, as the repair computation is done by computing the vertex cover of the interdependency graph. A bigger number of invalid data assertions leads to a bigger graph and this leads to a more timely computation of the vertex cover.

Another significant impact factor of the repair computation time is the amount of interdependencies in the interdependency graph. We can see that the repair computation time increases with a higher rate in Figure 6.4 than in Figure 6.5, which can be explained by the fact that the DBpedia 3.6 ontology contains only functional constraints, which form cliques in the interdependency graph (thus, more interdependencies), whereas the DBpedia 3.9 ontology contains mainly disjointness constraints, which cause less interdependencies, therefore less edges in the interdependency graph.

Moreover, the repair application time seems to be negligible in all of the above experiments. This is due to the fact that the repair application is performed by executing a single SPARQL-Update query containing the deletion of all the triples in the repairing delta, which is very efficient due to the optimizations for batch operations performed by Virtuoso.

The last significant conclusion from the three first sets of experiments comes from the comparison of the times measured for the two different DBpedia ontology versions. We can see that the diagnosis times using the version 3.9 of the DBpedia ontology are two orders of magnitude higher compared to the diagnosis times measured using the version 3.6. This is due to the fact that the $cln(\mathcal{T})$ of version 3.9 contains 323415 constraints, whereas the one of version 3.6 contains only 18. The bigger number of constraints of version 3.9 leads to the generation of more queries to be executed by the diagnosis algorithm, causing this big difference in the measurements.

4th set of experiments

As a last set of experiments, we combined our framework with the approach evaluated in [1]. We put an effort in recreating the datasets presented in that work, but we didn't manage to achieve that, as the datasets that are available in the webpage

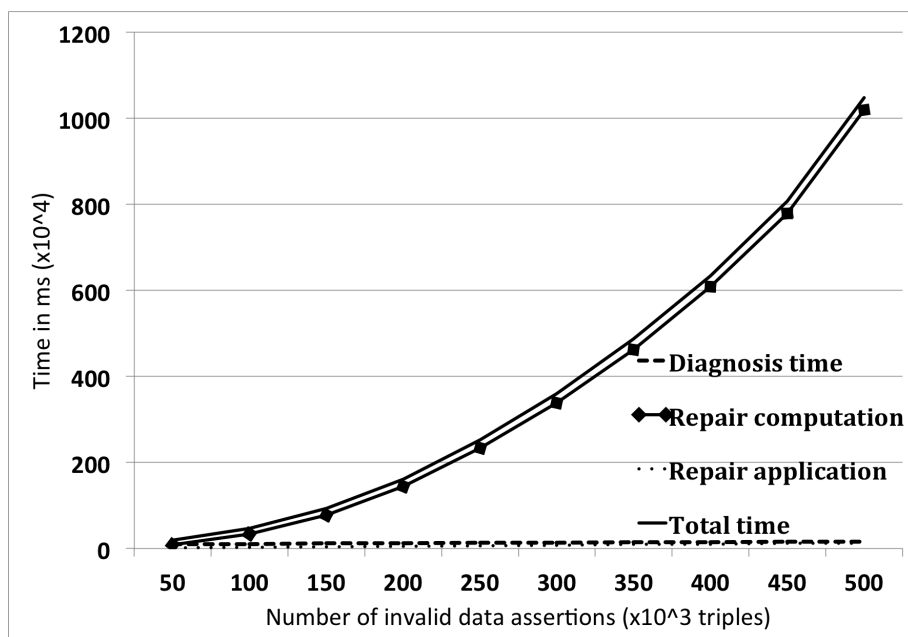


Figure 6.4: Performance for DBpedia ontology version 3.6 with datasets of varying number of invalid data assertions and fixed dataset size (10M triples).

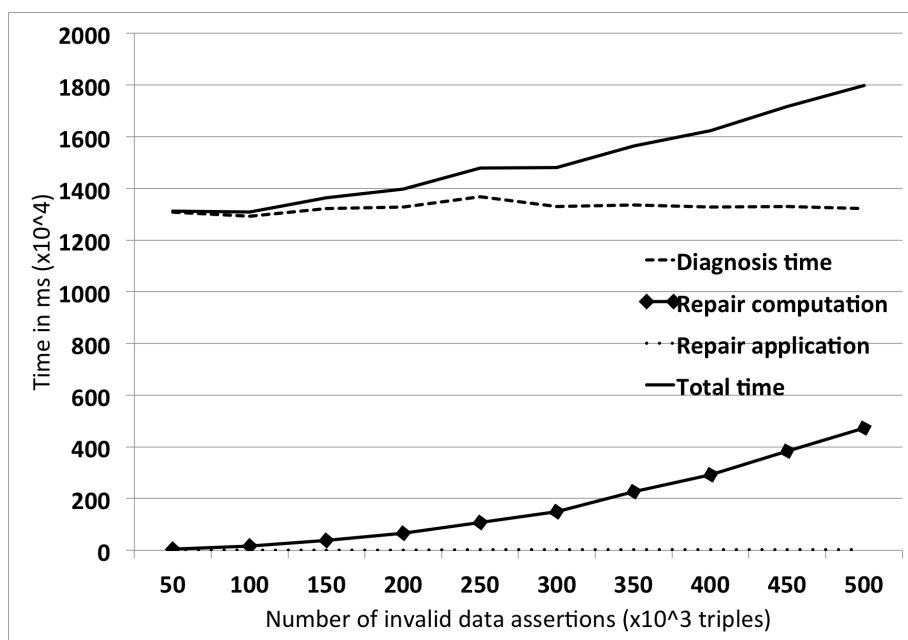


Figure 6.5: Performance for DBpedia ontology version 3.9 with datasets of varying number of invalid data assertions and fixed dataset size (10M triples).

of that evaluation⁴ are given as PostgreSQL dump files and we could not load them as RDF triples. However, we managed to produce RDF files that had the same characteristics (percentage of invalidities, uniform coverage of the constraints of the extended LUBM ontology) as the datasets used in [1], by using our synthetic data generator. Note that sizes of the datasets have some differences, which, however, are minor.

After producing the datasets for this set of experiments, we evaluated the performance of our framework, using as input the combination of the extended LUBM ontology and the datasets produced (16 measurements). At first, we used the greedy approach for the computation of the vertex cover (i.e., the repairing delta), and the results of this evaluation (in comparison with the total times presented in [1]) can be found in Figure 6.6.

As it was noted in Chapter 2, the work in [1] computes the repairing delta in a different way than we do in our framework, resolving each invalidity by removing both data assertions that take part in it and, in this way, removing more information than necessary. The approach in that work is, essentially, the same as computing the vertex cover using the 2-approximation algorithm [25] in the context of our framework. For this reason, we also evaluated our framework with the use of the 2-approximation algorithm for the computation of the vertex cover. The results of this evaluation can also be found in Figure 6.6.

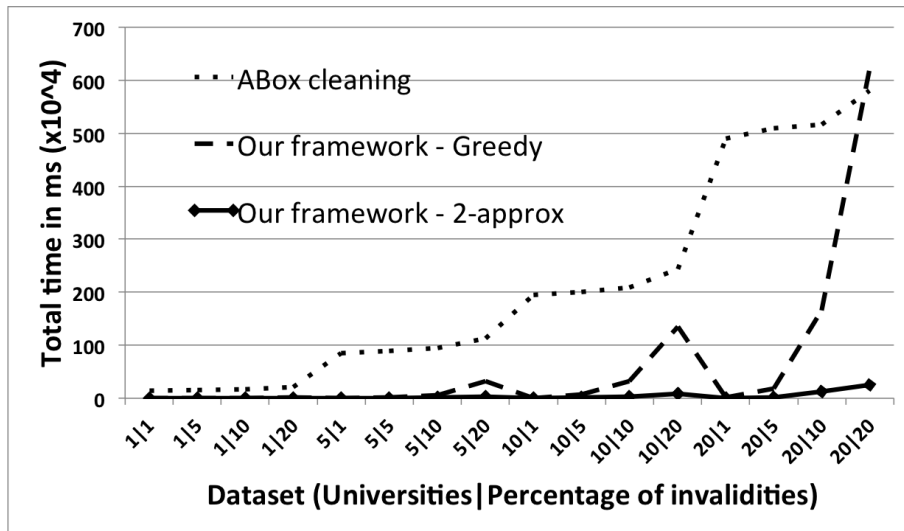


Figure 6.6: Comparison of our framework with the approach evaluated in [1].

From the results of this last set of experiments, we can conclude that our framework performs better than the approach in [1] in the vast majority of the experiments, for both the greedy computation of the vertex cover and the 2-approximation approach. The only experiment in which our framework performed

⁴<http://www.dis.uniroma1.it/~ruzzi/quid>

worse is the experiment with the dataset of 20 universities and 20% of invalidities, with the use of the greedy computation. This can be explained by the fact that the greedy computation calculates the vertex with the greatest degree in each step of the process, which can be a time-consuming task. It can be seen by the comparison of their evaluation, that the greedy computation is far more complex than the 2-approximation, thus giving worse times overall. However, in this way the repair that is produced removes less information from the original dataset, thus it is of better quality. This is also illustrated in Table 6.5, which gives a comparison of the repairing deltas produced using the greedy computation and the 2-approximation.

Dataset	Repairing delta (# of triples)	
	Greedy computation	2-approximation
1 University/1% invalidities	349	654
1 University/5% invalidities	1628	3226
1 University/10% invalidities	2376	4752
1 University/20% invalidities	10559	18564
5 Universities/1% invalidities	1957	3610
5 Universities/5% invalidities	8750	17114
5 Universities/10% invalidities	19610	38398
5 Universities/20% invalidities	46719	89116
10 Universities/1% invalidities	2681	5176
10 Universities/5% invalidities	25817	47426
10 Universities/10% invalidities	44983	86350
10 Universities/20% invalidities	77831	145792
20 Universities/1% invalidities	8986	16886
20 Universities/5% invalidities	25582	50584
20 Universities/10% invalidities	97186	186462
20 Universities/20% invalidities	176044	352088

Table 6.5: Comparison of the repairing deltas produced by using the greedy computation of the vertex cover and the 2-approximation, in the 4th set of experiments

6.6 Conclusions from the evaluation

Summing up the outcome of the experimental evaluation, we can come to the following main conclusions:

- Diagnosis, in the context of our framework, can be performed in linear time with respect to the size of the dataset.
- Repair computation can be performed in polynomial time with respect to the number of invalid data assertions that appear in the dataset.

- Our implementation enjoys a decent performance in real-world settings, with large datasets, large numbers of constraints and invalidities, being able to repair the huge DBpedia 3.9 (>2B triples) in about 11 hours, which is a reasonable amount of time given that repairing is expected to be an offline process. This fact also proves that our framework can be used on top of already deployed knowledge bases, without any further reconfiguration.
- In terms of performance and quality of the produced repair, our framework compares well to the only related work addressing the automatic repairing of an inconsistent *DL-Lite_A* KB [19].

Algorithm 6 RandomWalk($v, s, o, INV, d, dist, n$)

Input: A vertex v of the $TPG = (V, E)$, a subject URI s , and object URI o , the set of already produced invalidities INV and the distance covered so far d

```

1: if  $d \leq dist$  then
2:    $neighbors$  = a random number of neighbors of  $v$ 
3:   for  $neighbors$  do
4:     while  $|INV| < n$  do
5:       Randomly pick a vertex  $v'$  from the neighbors of  $v$ 
6:       Create a new triple  $t$ 
7:        $subject(t) = subject(v')$ 
8:        $predicate(t) = predicate(v')$ 
9:        $object(t) = object(v')$ 
10:      if SubjectWithSubject $\in$ Edge( $v, v'$ ) then
11:         $subject(t) = s$ 
12:      else if ObjectWithSubject $\in$ Edge( $v, v'$ ) then
13:         $subject(t) = o$ 
14:      else
15:         $subject(t) = \text{NewResourceURI}$ 
16:      end if
17:      if ObjectWithObject $\in$ Edge( $v, v'$ ) then
18:         $object(t) = o$ 
19:      else if SubjectWithObject $\in$ Edge( $v, v'$ ) then
20:         $object(t) = s$ 
21:      else
22:         $object(t) = \text{NewResourceURI}$ 
23:      end if
24:       $INV \leftarrow INV \cup t$ 
25:      RandomWalk( $v', subject(t), object(t), INV, d + 1, dist, n$ )
26:    end while
27:  end for
28: end if

```

Chapter 7

Conclusions and future work

7.1 Conclusions

In this work, we presented a novel, fully automatic and modular diagnosis and repairing framework for assisting KB curators in the arduous task of enforcing integrity constraints in large datasets, taking into account logical inference, and maintaining consistency of the KB. The framework we have presented is based on the *DL-Lite_A* ontology language, with which we can express and diagnose several useful types of logical constraints, while maintaining good computational features. Our framework also uses a novel graph representation of the invalidities in the KB, which gives us the ability to use notions of graph theory to lead the repairing process.

We have implemented our framework and produced two different applications, one running from the console and one with a web interface. The experimental evaluation of our framework implementation shows that it can be used on top of already deployed KBs without any further reconfiguration. Moreover, the evaluation shows that our framework is scalable for large dataset sizes, often found in real reference KBs, such as DBpedia. More specifically, it can handle datasets of billions of triples and large numbers of invalidities (millions of triples).

7.2 Future work

As a direction for our future work, we aim to enrich our framework beyond the fully automatic approach. We are considering a semi-automatic approach, which will be guided by user guidelines/preferences over the possible resolutions of invalidities, thus providing a repair that is as close as possible to the user needs.

We are also considering a manual approach, which will allow the user to perform massive repairing of several violations at once, with a few clicks, rendering the manual repairing of millions of invalidities possible. This approach will also be based on the interdependency graph to lead the decisions of the user.

As another direction, we plan to consider improvements on the scalability properties of our algorithms. More precisely, we plan to consider different algorithms, and their computational properties, for the computation of the vertex cover, as this can be a very time-consuming task, as shown by the experimental evaluation. Moreover, we will be considering different techniques for querying the triple store for invalidities, possibly by combining similar queries and, in this way, executing less, but more complex, queries. We are also considering improvements on the implemented web application, in the direction of stability and better functionality.

Last, as a direction of future research and extension of our work, we plan to incorporate a TBox diagnosis and repair component in our framework, in order to be able to check if the TBox is satisfiable and, if not, in what manner it could be repaired. Moreover, we are considering the changes that would be necessary in our algorithms and framework, in order to be able to capture incremental data updates in the ABox and to perform incremental diagnosis and repairing, without having to check over the entire ABox.

Bibliography

- [1] R. Rosati, M. Ruzzi, M. Graziosi, and G. Masotti, “Evaluation of techniques for inconsistency handling in OWL 2 QL ontologies,” in *The Semantic Web—ISWC 2012*. Springer, 2012, pp. 337–349.
- [2] C. Bizer, T. Heath, and T. Berners-Lee, “Linked data—the story so far,” *International journal on semantic web and information systems*, vol. 5, no. 3, pp. 1–22, 2009.
- [3] B. Motik, I. Horrocks, and U. Sattler, “Bridging the gap between OWL and relational databases,” *Web Semantics: Science, Services and Agents on the World Wide Web*, vol. 7, no. 2, pp. 74–89, 2009.
- [4] J. Tao, E. Sirin, J. Bao, and D. L. McGuinness, “Integrity constraints in OWL.” in *AAAI*, 2010.
- [5] A. Deutsch, “Fol modeling of integrity constraints (dependencies),” in *Encyclopedia of Database Systems*. Springer, 2009, pp. 1155–1161.
- [6] D. Kontokostas, P. Westphal, S. Auer, S. Hellmann, J. Lehmann, and R. Cornelissen, “Databugger: a test-driven framework for debugging the web of data,” in *Proceedings of the companion publication of the 23rd international conference on World wide web companion*. International World Wide Web Conferences Steering Committee, 2014, pp. 115–118.
- [7] F. N. Afrati and P. G. Kolaitis, “Repair checking in inconsistent databases: algorithms and complexity,” in *Proceedings of the 12th International Conference on Database Theory*. ACM, 2009, pp. 31–41.
- [8] J. Chomicki and J. Marcinkowski, “On the computational complexity of minimal-change integrity maintenance in relational databases,” in *Inconsistency Tolerance*. Springer, 2005, pp. 119–150.
- [9] D. Calvanese, G. De Giacomo, D. Lembo, M. Lenzerini, A. Poggi, and R. Rosati, “Linking data to ontologies: The Description Logic *DL-Lite_A*.” in *OWLED*, 2006.

- [10] M. Arenas, L. Bertossi, and J. Chomicki, “Consistent query answers in inconsistent databases,” in *Proceedings of the eighteenth ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*. ACM, 1999, pp. 68–79.
- [11] G. Greco, S. Greco, and E. Zumpano, “A logical framework for querying and repairing inconsistent databases,” *Knowledge and Data Engineering, IEEE Transactions on*, vol. 15, no. 6, pp. 1389–1408, 2003.
- [12] J. Chomicki and J. Marcinkowski, “Minimal-change integrity maintenance using tuple deletions,” *Information and Computation*, vol. 197, no. 1, pp. 90–121, 2005.
- [13] M. Arenas, L. Bertossi, and J. Chomicki, “Scalar aggregation in fd-inconsistent databases,” in *Database Theory—ICDT 2001*. Springer, 2001, pp. 39–53.
- [14] L. Grieco, D. Lembo, R. Rosati, and M. Ruzzi, “Consistent query answering under key and exclusion dependencies: Algorithms and experiments,” in *Proceedings of the 14th ACM international conference on Information and knowledge management*. ACM, 2005, pp. 792–799.
- [15] J. Wijsen, “Consistent query answering under primary keys: a characterization of tractable queries,” in *Proceedings of the 12th International Conference on Database Theory*. ACM, 2009, pp. 42–52.
- [16] G. Lausen, M. Meier, and M. Schmidt, “SPARQLing constraints for RDF,” in *Proceedings of the 11th international conference on Extending database technology: Advances in database technology*. ACM, 2008, pp. 499–509.
- [17] D. Lembo, M. Lenzerini, R. Rosati, M. Ruzzi, and D. F. Savo, “Query rewriting for inconsistent DL-Lite ontologies,” in *Web Reasoning and Rule Systems*. Springer, 2011, pp. 155–169.
- [18] A. Acciarri, D. Calvanese, G. De Giacomo, D. Lembo, M. Lenzerini, M. Palmieri, and R. Rosati, “QUONTO: querying ontologies,” in *AAAI*, vol. 5, 2005, pp. 1670–1671.
- [19] G. Masotti, R. Rosati, and M. Ruzzi, “Practical ABox cleaning in DL-Lite (progress report).” in *Description Logics*, 2011.
- [20] F. Baader, *The description logic handbook: theory, implementation, and applications*. Cambridge university press, 2003.
- [21] D. Calvanese, G. De Giacomo, D. Lembo, M. Lenzerini, and R. Rosati, “Tractable reasoning and efficient query answering in description logics: The DL-Lite family,” *Journal of Automated Reasoning*, vol. 39, no. 3, pp. 385–429, 2007.

- [22] D. Calvanese, G. De Giacomo, D. Lembo, M. Lenzerini, A. Poggi, M. Rodriguez-Muro, and R. Rosati, “Ontologies and databases: The DL-Lite approach,” in *Reasoning Web. Semantic Technologies for Information Systems*. Springer, 2009, pp. 255–356.
- [23] S. Abiteboul, R. Hull, and V. Vianu, *Foundations of databases*. Addison-Wesley Reading, 1995, vol. 8.
- [24] M. R. Garey and D. S. Johnson, *Computers and intractability*. Freeman San Francisco, 1979, vol. 174.
- [25] C. H. Papadimitriou and K. Steiglitz, *Combinatorial optimization: algorithms and complexity*. Courier Dover Publications, 1998.
- [26] G. Karakostas, “A better approximation ratio for the vertex cover problem,” in *Automata, languages and programming*. Springer, 2005, pp. 1043–1050.
- [27] C. Bizer and A. Schultz, “Berlin SPARQL benchmark (BSBM) Experiment - v3.1,” 2013.