International Conference on Logic Programming ICLP 2008

Udine, Italy

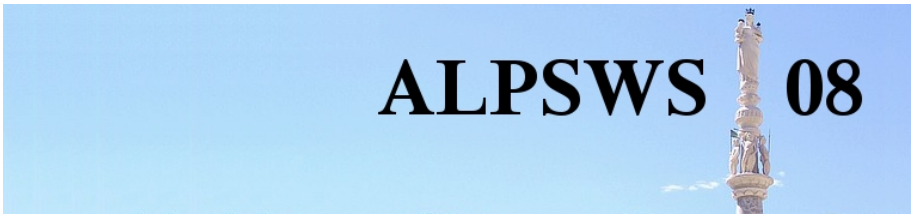
9-13 December 2008

ICLP 2008 Workshop

ALPSWS 2008: Applications of Logic Programming to the (Semantic) Web and Web Services

12 December 2008

Proceedings



Editors:

J. de Bruijn, S. Heymans, D. Pearce, A. Polleres, and E. Ruckhaus

Preface

This volume contains the papers presented at the third international workshop on *Applications of Logic Programming to the (Semantic) Web and Web Services (ALPSWS2008)* held on the 12h of December 2008 in Udine, Italy, as part of the 24$^{\text{th}}$ International Conference on Logic Programming (ICLP 2008).

The advent of the Semantic Web promises machine readable semantics and a machine-processable next generation of the Web. The first step in this direction is the annotation of static data on the Web by machine processable information about knowledge and its structure by means of Ontologies. The next step in this direction is the annotation of dynamic applications and services invocable over the Web in order to facilitate automation of discovery, selection and composition of semantically described services and data sources on the Web by intelligent methods; this is called Semantic Web Services.

Many workshops and conferences have been dedicated to these promising areas, mostly covering generic topics. The ALPSWS workshop series has a slightly different goal. Rather that bringing together people from a wide variety of research fields with different understandings of the topic, we have tried to focus on the various application areas and approaches in this domain from the perspective of declarative logic programming (LP).

The workshop provides a snapshot of the state of the art of the applications of LP to the Semantic Web and to Semantic Web Services, with the following main objectives and benefits:

- Bring together people from different sub-disciplines of LP to focus on technological solutions and applications from LP to the problems of the Web.
- Promote further research in this interesting application field.

The 2008 edition of ALPSWS includes work on the topic of *integrating ontologies and rules*, but also integration with machine learning. Furthermore, we can see an interest in *integration with database technology*, a prerequisite for large-scale adoption of Semantic Web technology. Then, two important challenges in reasoning on the Web are addressed, namely combining open- and closed-world reasoning and *reasoning with large data sets*. Finally, there is an application of logic programming to service description and drug discovery.

November 2008 The Editors

Workshop Organization

Organizing Committee

Jos de Bruijn
Stijn Heymans
Axel Polleres
David Pearce
Edna Ruckhaus

Programme Committee

Carlos Damasio
Thomas Eiter
Cristina Feier
Gopal Gupta
Claudio Gutierrez
Giovambattista Ianni
Uwe Keller
Markus Kroetzsch
Zoe Lacroix
Gergely Lukácsy
Wolfgang May
Enrico Pontelli
Hans Tompits
Alejandro Vaisman
Maria Esther Vidal
Gerd Wagner

Additional Reviewers

Aidan Hogan
Thomas Krennwallner
Francesco Ricca
Mantas Simkus

Table of Contents

Full Papers

Short Paper

Upgrading Databases to Ontologies*

Gisella Bennardo, Giovanni Grasso, Nicola Leone, Francesco Ricca

Department of Mathematics, University of Calabria, 87036 Rende (CS), Italy
{lastname}@mat.unical.it

Abstract. In this paper we propose a solution that combines the advantages of an ontology specification language, having powerful rule-based reasoning capabilities, with the possibility to efficiently exploit large (and, often already existent) enterprise databases. In particular, we allow to "upgrade" existing databases to an ontology for building a unified view of the enterprise information. Databases are kept and the existing applications can still work on them, but the user can benefit of the new ontological view of the data, and exploit powerful reasoning and information integration services, including: problem-solving, consistency checking, and consistent query answering. Importantly, powerful rule-based reasoning can be carried out in mass-memory allowing to deal also with data-intensive applications.

Kewwords: Ontologies, Rules, Databases, Answer Set Programming, Information Integration, Consistent Query Answering.

1 Introduction

In the last few years, the need for knowledge-based technologies is emerging in several application areas and, in particular, both enterprises and large organizations are looking for powerful instruments for knowledge-representation and reasoning. In this field, ontologies [1] have been recognized to be a fundamental tool. Indeed, they are well-suited formal tools that provide both a clean abstract model of a given domain and powerful reasoning capabilities. In particular, they have been recently exploited for specifying terms and definitions relevant to business enterprises, obtaining the so-called *enterprise/corporate ontologies*. Enterprise/Corporate ontologies can be used to share/manipulate the information already present in a company; in fact, they provide for a "conceptual view" expressing at the intensional level complex relationships among the entities of enterprise domains. In this way, they can offer a convenient access to the enterprise knowledge, simplifying the retrieval of information and the discovery of new knowledge through powerful reasoning mechanisms. However, enterprise ontologies are not widely used yet, mainly because of two major obstacles: (*i*) the specification of a real-world enterprise ontology is an hard task; and, (*ii*) usually, enterprises already store their relevant information in large databases. As far as point (*i*) is concerned, it can be easily seen that developing an enterprise ontology by scratch would be a time-consuming and expensive task, requiring the cooperation of knowledge engineers with domain

* Supported by M.I.U.R. within projects "Potenziamento e Applicazioni della Programmazione Logica Disgiuntiva" and "Sistemi basati sulla logica per la rappresentazione di conoscenza: estensioni e tecniche di ottimizzazione."

experts. Moreover, (*ii*) the obtained specification must incorporate the knowledge (mainly regarding concept instances) already present in the enterprise information systems. This knowledge is often stored in large (relational) database systems, and loading it again in the ontologies may be unpractical or even unfeasible. This happens because of the large amount of data to deal with, but also since databases have to keep their autonomy (considering that many applications work on them). In addition, when data residing in several autonomous sources are combined in a unified view, *inconsistency problems* may arise [2, 12] that cannot be easily fixed.

In this paper we describe a solution that combines the advantages of an ontology representation language (i.e., high expressive power and clean representation of data) having powerful rule-based reasoning features, with the capability to efficiently exploit large (and, often already existent) enterprise databases. Basically, if we are given some existing databases, we can analyze their schema and try to recognize both entities and relationships they store. This information is exploited for "upgrading" the database to an ontology. Here, ontology instances are "virtually" specified (i.e. they are linked, not imported) by means of special logic rules which define a mapping from the data in the database to the ontology. The result is a unified ontological specification of the enterprise information that can be employed, for browsing, editing and advanced reasoning. Moreover, possible inconsistent information obtained by merging several databases is dealt with by adopting data-integration techniques.

We developed these solutions in OntoDLV [3–5], a system that implements a powerful logic-based ontology representation language, called OntoDLP, which is an extension of (disjunctive) Answer Set Programming [6–8] (ASP) with all the main ontology constructs including classes, inheritance, relations, and axioms.[1] OntoDLP combines in a natural way the modeling power of ontologies with a powerful "rule-based" language allowing for disjunction in rule heads and nonmonotonic negation in rule bodies. In general, disjunctive ASP, and thus OntoDLP, can represent *every* problem in the complexity class Σ_2^{P} and Π_2^{P} (under brave and cautious reasoning, respectively) [9].

Summarizing, the main contributions of this paper are:

- an extension of OntoDLP by suitable constructs, called *virtual class* and *virtual relation*, which allows one to specify the extensions of ontology concepts/relations by using data from existing relational databases;
- the design of a rewriting technique for implementing Consistent Query Answering (CQA) [2, 10–13] in OntoDLV. CQA allows for obtaining as much consistent information as possible from queries, in case of global inconsistent information.

Moreover, we efficiently implemented the proposed extensions in the OntoDLV system by allowing for the evaluation of queries in mass memory. In this way, OntoDLV can seamlessly provide to the users both an integrated ontological view of the enterprise knowledge and efficient query processing on existing data sources.

[1] The term "Answer Set Programming" was introduced by Vladimir Lifschitz in his invited talk at ICLP'99 to denote the declarative programming paradigm originally described in [6]. Since ASP is the most prominent branch of logic programming in which rule heads may be disjunctive, the term Disjunctive Logic Programming (DLP) refers explicitly to ASP. OntoDLP takes its name from ontologies plus DLP.

2 The OntoDLP language

In this section we briefly overview OntoDLP, an ontology representation and reasoning language which provides the most important ontological constructs and combines them with the reasoning capabilities of ASP. For space limitations we cannot include a detailed description of the language. The reader is referred to [4, 5] for details. Moreover, hereafter we assume the reader to be familiar with ASP syntax and semantics, for further details refer to [6, 14].

More in detail, the OntoDLP language includes, the most common ontology constructs, such as: **classes**, **relations**, (multiple) **inheritance**; and the concept of modular programming by means of **reasoning modules**. A *class* can be thought of as a collection of individuals. An individual, or *object*, is any identifiable entity in the universe of discourse. Objects, also called class instances, are unambiguously identified by their object-identifier (oid) and belong to a class. A class is defined by a name (which is unique) and an ordered list of typed attributes, identifying the properties of its instances. Classes can be organized in a specialization hierarchy (or data-type taxonomy) using the built-in *is-a* relation (*multiple inheritance*). The following are examples of both class and instance declarations:

$$\textbf{\textit{class}}\ person(name:string, father:person, mother:person, birthplace:place).$$
$$\textbf{\textit{class}}\ employee\ \ \textbf{\textit{isa}}\ \{person\}(salary:integer, boss:person).$$
$$john:person(name:\text{“}John\text{”}, father:jack, mother:ann, birthplace:rome).$$

Relationships among objects are represented by means of *relations*, which, like classes, are defined by a (unique) name and an ordered list of attributes. As in ASP, logic programs are sets of logic rules and constraints. However, OntoDLP extends the definition of logic atom by introducing class and relation predicates, and complex terms (allowing for a direct access to object properties). Logic rules can be exploited for defining classes and relations when their instances can be “derived” (or inferred) from the information already stated in an ontology. This kind of intensional constructs are called *Collection classes* and *Intensional Relations*. Basically, collection classes *collect* instances defined by another class and perform a re-classification based on some information which is already present in the ontology; whereas, intentional relations are similar to (but more powerful of) database views. Importantly, the programs (set of rules) defining collection classes (and intensional relations) must be normal and stratified (see e.g., [15]). For instance, the class *richEmployee* can be defined as follows:

$$\textbf{\textit{collection class}}\ richEmployee(name:string)\{$$
$$E:richEmployee(name:N) \text{ :- } E:employee(name:N, salary:S), S > 1000000.\}$$

Moreover, OntoDLP allows for special logic expressions called *axioms* modeling sentences that are always true. Axioms provide a powerful mean for defining/checking consistency of the specification (i.e., discard ontologies which are, somehow, contradictory or not compliant with the domain's intended perception). For example, we may enforce that a person cannot be father of himself by writing: $\text{:- } X : person(father:X)$.

In addition to the ontology specification, OntoDLP provides powerful reasoning and querying capabilities by means of the language components *reasoning modules* and *queries*. In practice, a *reasoning module* is a disjunctive ASP program conceived to reason about the data described in an ontology. Reasoning modules are identified by a name and are defined by a set of (possibly disjunctive) logic rules and integrity

constraints; clearly, the rules of a module can access the information present in the ontology.

An important feature of the language is the possibility of asking conjunctive queries, that, in general, can involve both ontology entities and reasoning modules predicates. As an example, we ask for persons whose father is born in Rome as follows: $X : person(father : person(birthplace : place(name : \text{“}Rome\text{”})))$?

3 Virtual Classes and Virtual Relations

In this section we show how an existing database can be “upgraded” to an OntoDLP ontology. In particular, the new features of the language, called *virtual classes* and *virtual relations*, are described by exploiting the following example.

Suppose that a Banking Enterprise asks for building an ontology of its domain of interest. This request has the goal of obtaining a uniform view of the knowledge stored in the enterprise information system that is shared among all the enterprise branches.

Table	Attributes
Branch	branch-name, branch-city, assets
Customer	customer-name, social-security, customer-street, customer-city
Depositor	customer-social-sec , account-number, access-date
Saving-account	account-number, balance, interest-rate
Checking-account	account-number, balance,overdraft-amount
Loan	loan-number , amount, branch-name
Borrower	customer-social-sec, loan-number
Payment	loan-number , payment-number, payment-date, payment-amount

Table 1. The Banking Enterprise Database.

The schema of the existing database of the enterprise is reported in Table 1. The first step that must be done is to reconstruct the semantics of the data stored in this database. It is worth noting that, in general, a database schema is the product of a previously-done modeling step on the domain of interest. Usually, the result of this conceptual-design phase is a semantic data model that describes the structure of the entities stored in the database. Likely, the database engineers exploited the Entity-Relationship Model (ER-model) [17], that consists of a set of basic objects (called entities), and of relationships among these objects. The ER-model underlying a database can be reconstructed by reverse-engineering[2] or can be directly obtained from the documentation of the original project.

Suppose now that, we obtained the ER-model corresponding to the database of Table 1. In particular, the corresponding ER diagram is shown in Figure 1. From this diagram it is easy to recognize that the enterprise is organized into *branches*, which are located into a given place and also have an asset and a unique name. A bank *customer* is identified by its social-security number and, in addition, the bank stores information about customer's name, street and living place. Moreover, customers may have *accounts* and can take out *loans*. The bank offers two types of *accounts*: *saving-accounts* with an interest-rate, and *checking-accounts* with a overdraft-amount. To each account is assigned a unique account-number, and maintains last access date. Moreover, accounts can be held by more than one

[2] Note that, the reverse-engineering task is not trivial, and even automatic methods may fail to reconstruct the original semantics [18].

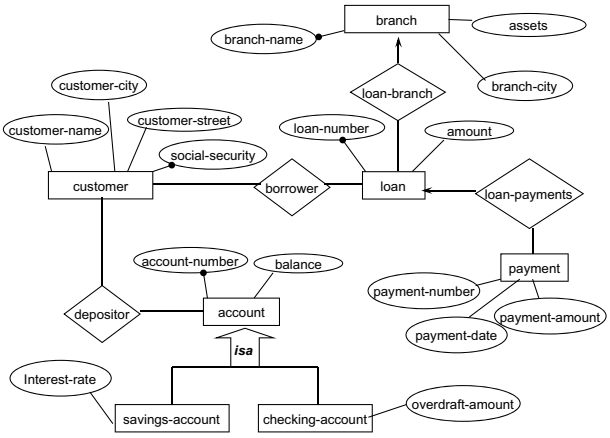


Fig. 1. The Banking Enterprise ER diagram

customer, and obviously one customer can have various accounts (*depositors*). Note that, in the case of *accounts*, the ER-model exploits specialization/generalization construct. A *loan* is identified by a unique loan-number and, as well as accounts, can be held by several customers (*borrowers*). In addition, the bank keeps track of the loan amount and *payments* and also of the branch at the loan originates. For each payment the bank records the date and the amount; for a specific loan a payment-number univocally identifies a particular payment.

All this information represents a good starting point for defining an ontology that describes the banking enterprise domain.[3] As a matter of fact, we can easily exploit it both for identifying ontology concepts and for detecting the database tables which store data about ontology instances. In practice, we can "upgrade" the banking database to a banking ontology by creating an OntoDLP (base) class, with name c, for each concept c in the domain; and by exploiting logic rules that specify a mapping between class c and its instances "stored" in the database. A class c defined by means of mapping rules is called *virtual*, because its instances come from an external source; but, as far as reasoning and querying are concerned they are like any other class directly specified in OntoDLP. More in detail, a *virtual class* is defined by using the keywords **virtual class** followed by the class name, and by the specification of class attributes; then, instances are defined by means of rules containing special atoms that allows for accessing the source database.

First of all, external data sources are specified directly in OntoDLP, as instances of the built-in class *dbSource* as follows:

$$db1 : dbSource(connectionURI : \text{“}http://db.banking.com\text{”}, user : \text{“}myUser\text{”}, password : \text{“}myPsw\text{”}).$$

Here, the object identifier *db1* is used to identify the enterprise database. Note that such a mechanism allows to build an ontology starting from one or more databases, just specifying more *dbSources*; moreover, this source identification strategy is sufficiently general to be (in the future) extended also to access other kind of sources

[3] Note also that, our goal is not to provide a tool for reasoning on ER schemata; instead, we allow the ontology engineer to design and "populate" an ontology that exploits data about the *instances* that is stored in relational databases.

beside databases. Now, given the source identifier for the enterprise database, we model the *branch* entity as follows:

$$\textbf{\textit{virtual class}}\ branch(name:string, city:string, assets:integer)\{$$
$$f(BN) : branch(name:BN, city:BC, assets:A) :-$$
$$branch@db1(branch\text{-}name:BN, branch\text{-}city:BC, assets:A).\}$$

The rule acts as mapping between the data contained in table *branch* and the instances of class *branch* by exploiting a new type of atom, called *sourced atom*. A *sourced atoms* consist of a name (*branch*), that identifies a table "at" (@) a specific database source (*db1*), and a list of attributes (that match the table schema). Attributes can be filled in by constants or variables.

Note that, whereas databases store values, ontologies manage instances (which are not values) that are uniquely identified by oids.[4] We provided a specific solution for facing with this problem, in which values appearing in the databases are kept, someway, distinct from object identifiers appearing in the ontology. In particular, *functional object identifiers*, suitably built from database values, are exploited for identifying ontology instaces. In our example, the head of the mapping rule contains the functional term $f(BN)$, that builds, for each instance of *branch*, a *functional object identifier* composed of the functor f containing the value of the *name* attribute stored in the table *branch*. In practice, if the *branch* table stores a tuple *("Spagna","Rome", 1000000)*, then the associated instance in the ontology will be: $f(\text{"}Spagna\text{"}) : branch(name:\text{"}Spagna\text{"}, city:\text{"}Rome\text{"}, assets:1000000)$. In this way, the *functional object identifier* $f(\text{"}Spagna\text{"})$ is built from the data value *"Spagna"*, keeping the data alphabet distinct from the one of *object identifiers*.

Note that *name* is a key for table *branch*. Because object identifiers in OntoDLP uniquely identify instances, it is preferable to exploit only keys for defining functional object identifiers. This simple policy ensures that we will obtain an admissible ontology whenever the source database is unique and consistent; whereas, if more than one source database is exploited for defining ontology entities, some admissibility constraint for the ontology schema (like e.g. referential integrity constraints, unicity of object identifiers, etc. see [3])) might be violated. To face with this problem our system supports data integration features which are described in Section 4. Clearly, in order to ensure the maximum flexibility, the responsibility of writing a "right" ontology mapping is left to the ontology engineer.

We say that a *virtual class* declared by means of *sourced atoms* is in *logical notation*. We provided also an alternative notation for accessing database tables, called *SQL notation*. In particular, the *virtual class branch* can be equivalently defined as follows:

$$\textbf{\textit{virtual class}}\ branch(name:string, city:string, assets:integer)\{$$
$$f(BN) : branch(name:BN, city:BC, assets:A) :-$$
$$[db1, \text{"}SELECT\ branch\text{-}name\ AS\ BN, branch\text{-}city\ AS\ BC, assets\ AS\ A$$
$$FROM\ branch\text{"}]\}$$

Here, a special atom which contains an SQL query is used in the place of a sourced one. Formally, a *SQL atom* consists of a pair [db object identifier, sql query] enclosed in square brackets. The db object identifier picks out the database on which the sql query will be performed.

Consider now the *customer* entity. Also here, we define a virtual class as follows:

[4] This is the well-known impedance mismatch problem [19, 20].

$$\textbf{\textit{virtual class}}\ customer(ssn:string, name:string, street:string, city:string)\{$$
$$c(SSN) : customer(ssn:SSN, name:N, street:S, city:C) :-$$
$$customer@db1(social\text{-}security:SSN, customer\text{-}name:N, customer\text{-}street:S,$$
$$customer\text{-}city:C).\}$$

The functional term $c(SSN)$ is used here in order to assign to each instance a suitable *functional object identifier* built on the *social-security* attribute value. Note that, a fresh functor is used for each virtual class. In this way, functional object identifiers belonging to different classes are kept distinct. In our example, the *customer* and the *branch* class instances are made disjoint by using functor f and c, respectively.

Following the same methodology, we define a virtual class for the *loan* entity:

$$\textbf{\textit{virtual class}}\ loan(number:integer, loaner:branch, amount:integer)\{$$
$$l(N) : loan(number:N, loaner:f(L), amount:A) :-$$
$$loan@db1(loan\text{-}number:N, branch\text{-}name:L, amount:A).\}$$

Note that, the *loan* class has an attribute (*loaner*) of type *branch*. In this case, functional terms are carefully employed in order to maintain referential integrity. As shown above, the mapping uses the functional term $f(L)$ to build values for the *loaner* attribute. Basically, since the *branch* class use the functor f to build its object identifiers, then we also use the same functor where an object identifier of *branch* is expected.

In the following, we exploit the same idea to model the *payment* entity:

$$\textbf{\textit{virtual class}}\ payment(ref\text{-}loan:loan, number:integer, payDate:date,$$
$$amount:integer)\{$$
$$p(l(L), N) : payment(ref\text{-}loan:l(L), number:N, payDate:D, amount:A) :-$$
$$payment@db1(loan\text{-}number:L, payment\text{-}number:N, payment\text{-}date:D,$$
$$payment\text{-}amount:A).\}$$

Also in this case we deal with referential integrity constraints by using a proper functional term $l(L)$ where a *loan* object identifier is expected (*ref-loan* attribute); moreover, since payments are identified by a pair (payment-number, relaive loan) each instance of *payment* will be identified by a functional object identifier with two arguments: one of these is a functional object identifier of type *loan*; and, the other is the loan number.

As far as *accounts* are concerned, we know from the ER-model that they are specialized in two types: *saving-accounts* and *checking-accounts*. This situation can be easily dealt with by exploiting inheritance (see Section 2). Thus, we first define a *virtual class* named *account* as follows:

$$\textbf{\textit{virtual class}}\ account(number:integer, balance:integer).$$

and, then, we provide two *virtual classes*, *savingAccount* and *checkingAccount*, namely, which are declared to be both subclasses of *account*:

$$\textbf{\textit{virtual class}}\ savingAccount\ \ \textbf{\textit{isa}}\ \{account\}(interestRate:integer)\{$$
$$acc(N) : savingAccount(number:N, balance:B, interestRate:I) :-$$
$$saving\text{-}account\ @db1(account\text{-}number:N, balance:L, interest\text{-}rate:I).\}$$

virtual class $checkingAccount$ ***isa*** $\{account\}(overdraft : integer)\{$
$acc(N) : checkingAccount(number : N, balance : B, overdraft : I) :-$
$checking\text{-}account\ @db1(account\text{-}number : N, balance : L, overdraft\text{-}amount : I).\}$

In order to conclude our "upgrading" process, we have to model the relationships holding among the concepts in the banking domain. To deal with this problem, OntoDLP allows for defining also *virtual relations*. For instance, the ER diagram of Figure 1 shows that *customers* and *loans* are in relationship through *borrower* and *depositor*. Hence, we define two *virtual relations* as follows:

virtual relation $borrower(cust : customer, loan : loan)\{$
$borrower(cust : c(C), loan : l(L)) :-$
$borrower@db1(customer\text{-}social\text{-}sec : C, loan\text{-}number : L).\}$
virtual relation $depositor(cust : customer, account : account, , lastAccess : date)\{$
$depositor(cust : c(C), account : acc(A), lastAccess : D) :-$
$depositor@db1(customer\text{-}social\text{-}sec : C, account\text{-}number : A, access\text{-}date : d).\}$

It is worth noting that a *virtual relation* differs from a *virtual class* mainly because tuples are not equipped with object identifiers.

4 Data Integration Features

In previous sections we showed how a existing database can be upgraded to an OntoDLP ontology. Basically, the instances of ontology entities are virtually populated by means of special logic rules, which act as a mapping from the information stored in database tables to ontology instances. In general, the ontology engineer can obtain the data from several source databases, which are combined in a unified ontological view. This is a typical data integration scenario [2] where either some admissibility conditions on the ontology schema (e.g., referential integrity constraints, unicity of object identifiers, etc.), or some user-defined axioms might be violated by the obtained ontology.[5] In order to face with this problem, a possibility is to fix manually either the information in the sources or the ontology specification; but, if the ontology engineer can/does not want to modify the sources, then it would be very useful to single out as much *consistent* information as possible for answering queries. In our framework, we support both possibilities by offering the following data-integration features:

- *Consistency checking*: verify whether the obtained ontology is consistent or not, and, in the latter case, precisely detect tuples that violate integrity constraints or user defined axioms;
- *Consistent Query Answering (CQA)* [2, 10–13]: compute answer to queries that are true in every instance of the ontology that satisfies the constraints and differs minimally from the original one.

In the field of data-integration several notions of CQA have been proposed (see [12] for a survey), depending on whether the information in the database is assumed to be *correct* and *complete*. Basically, the incompleteness assumption coincides with

[5] It is easy to see that, our approach can be classified from a in data integration point of view as GAV (Global As View) [2] integration system.

the *open world assumption*, where facts missing from the database are not assumed to be false. Conversely, we assume that sources are complete. This choice, common in data warehousing, is suitable in a framework like OntoDLP that is based on the *closed world assumption*; and, as argued in [13], strengthen the notion of minimal distance from the original information.[6] There are two important consequences of this choice: integrity restoration can be obtained by only *deleting tuples* (note that the empty model is always a repair [13]); and, computing CQA for conjunctive queries remains *decidable* even when arbitrary sets of denial constraints and inclusion dependencies are employed [13].

More formally, given an OntoDLP ontology schema Σ and a set A of axioms or integrity constraints, let $\mathcal{O}$ and $\mathcal{O}^r$ be two ontology instances[7], we say that $\mathcal{O}^r$ is a *repair* [13] of $\mathcal{O}$ w.r.t. A, if $\mathcal{O}^r$ satisfies all the axioms in A and the instances in $\mathcal{O}^r$ are a maximal subset of the instances in $\mathcal{O}$. Basically, given a conjunctive query Q, *consistent answers* are those query results that are not affected by axioms violations and are true in any possible repair [13]. Thus, given and ontology instance $\mathcal{O}$ and a set of axioms A, a conjunctive query Q is consistently true in $\mathcal{O}$ w.r.t. A if Q is true in every repair of $\mathcal{O}$ w.r.t. A. Moreover, if Q is non-ground, the consistent answers to Q are all the tuples $\bar{t}$ such that the ground query $Q[\bar{t}]$ obtained by replacing the variables of Q by constants in $\bar{t}$ is consistently true in $\mathcal{O}$ w.r.t. A.

Note that, as shown in [13] the problem of computing consistent answers to queries (CQA) in the case of denial constraints and inclusion dependencies (such kind of constraints are suficient to model every admissibility condition on an OntoDLP schema[3, 4]) belongs to the Π_2^{P} complexity class; thus, they can be implemented by using disjunctive ASP.

In the next Section, we describe how the new features were implemented and in particular we show how to build an ASP program that implements CQA for the above mentioned kind of axtioms in the OntoDLV system.

5 Implementation

In this section, we first briefly describe the OntoDLV system [3]; and then, we detail the implementation of the new features, namely: *virtual classes/relations* and *consistent query answering*.

OntoDLV. OntoDLV is a complete framework that allows one to develop ontology-based applications. Thanks to a user-friendly visual environment, ontology engineers can create, modify, navigate, query ontologies, as well as perform advanced reasoning on them. An advanced persistency manager allows one to store ontologies transparently both in text files and internal relational databases; while powerful type-cheking routines are able to analyze ontology specifications and single out consistency problems. All the system features are made available to

[6] It is worth noting that, in relevant cases like denial constraints, query results coincide for both correct and complete information assumptions.

[7] Here ontology instance refers to the unique set of ground instances modeled by an ontology specification [3]. Note that, in our settingsOntoDLP axioms can model both denial constraints (like functional dependencies) and inclusion dependencies (in the latter case, negation as failure is exploited).

software developers trough an *Application Programming Interface* (API) that acts as a facade for supporting the development of applications based on OntoDLP [21]. The core of OntoDLV is a rewriting procedure (see [4]) that translates ontologies, axioms, reasoning modules and queries to an equivalent ASP program which, in the general case, runs on state-of-the art ASP system DLV [14]. Importantly, if the rewritten program is stratified and non disjunctive [6–8] (and the input ontology resides in relational databases) the evaluation is carried out directly in mass memory by exploiting a specialized version of the same system, called DLV^{DB} [22]. Note that, since entity specifications are stratified and non-disjunctive, queries on ontologies can always be evaluated in mass-memory (this is to say: "by exploiting a DBMS"). This makes the evaluation process very efficient , and allows the knowledge engineer to formulate queries in a language more expressive than SQL. Clearly, more complex reasoning tasks (whose complexity is NP/co-NP, and up to Σ_2^P/Π_2^P) are dealt with by exploiting the standard DLV system instead.

Virtual Classes and Virtual Relations. The implementation of *virtual classes* and *virtual relation* has been carried out by properly improving the rewriting procedure and by extending the persistency manager in order to provide both storage and manipulation facilities for virtual entities. More in detail, we implemented two different usage modalities: *off-line* and *on-line*.

In the first, the relevant information is extracted from the sources by exploiting SQL queries and, is stored into the internal data structures (basically, instances are "imported" and stored by exploiting the persistency manager). In the latter, queries are performed directly at the sources.

The *off-line* mode is preferable when one wants to migrate the database into an ontology, or when parts of a proprietary database are one-time granted to third parties. In fact, once the import is done, the source database can be disconnected, since instances are stored into the OntoDLV persistency manager. Obviously, depending on database size, the off-line modality could be time-consuming or even unpractical. In addition, one may want to keep the information in the original database (which is accessed by legacy applications), in order to deal with "fresh" information. In those cases, the *on-line* mode is preferable.

In both *on-line* and *off-line* modes, queries on the ontology are performed directly on mass-memory by exploiting DLV^{DB} [22]. To this end, we extended the rewriter procedure in such a way that DLV^{DB} mapping statements are properly generated. Indeed, DLV^{DB} takes as input both a logic program and a mapping specification linking database tables to logic predicates.

Importantly, in order to avoid the materialization of the entire ontology for evaluating an input query, an "unfolding" technique [2, 12] has also been integrated into the Rewriter module. Basically, when we have a query q on the ontology, every predicate of q is substituted with the corresponding query over the sources, provided that suitable syntactic conditions are satisfied.

As an example, if we ask for the instances of virtual class *branch* of Section 3 the following mapping directive for DLV^{DB} is generated by the rewriter procedure:

USEDB "*http* : *//db.banking.com*":myUser:myPsw.
USE *branch* (*branch-name*, *branch-city*, *assets*)
MAPTO *branchPredicate* (*varchar,varchar,integer*).

The above directive specifies the database (**USEDB**) on which the SQL query will be performed (may be the source database). Moreover, the listed attributes of the table *branch* (**USE**) are mapped (**MAPTO**) on the logic predicate *branch-Predicate*. In this case, *branchPredicate* is the predicate name used internally to rewrite in standard ASP the class branch.

Implementation of CQA. In order to implement consistent query answering we developed a new procedure in the OntoDLV system. Given an ontology $\mathcal{O}$, this procedure takes as input a conjunctive query Q, and a set of integrity constraints A and builds both an ASP program Π_{cqa} and a query Q_{cqa}, such that: Q is consistently true in $\mathcal{O}$ w.r.t. A iff Q_{cqa} is true in every answer set of Π_{cqa}, in symbols: $\Pi_{cqa} \models_c Q_{cqa}$ (in other words Q_{cqa} is cautious consequence of Π_{cqa}).

Note that, this can be done in our settings since CQA belongs to the Π_2^{P} complexity class [13]. However, we decided to support in the implementation only a family of constraints in such a way that complexity of CQA stays in co-NP. In particular, we consider constraints of the form:

$$(i) :- a_1(t_1), \cdots, a_n(t_n), \sigma(t_1, \ldots, t_n). \qquad (ii) :- a_1(t), not\ a_2(t).$$

where t_i is a tuple and $\sigma(t_1, \ldots, t_n)$ is a conjunction of comparison literals of the form $X\theta Y$, with $\theta \in \{<, >, =, \neq\}$ and X and Y are variables occurring in $t_1, \ldots, t_n$. In the database field constraints of type (i) are called *denial constraints*, whereas constraints of type (ii) allow for modeling *inclusion dependencies* (see [23]).[8] An inclusion dependency is often denoted by $Q[Y] \subseteq P[X]$ (where Q and P are relations) and it requires that all values of attribute Y in Q are also values of attribute X in some instance of P. For example, if P and Q are unary this can be ensured in OntoDLP by writing $:- Q(X), not\ P(X)$. In particular, we allow only acyclic[9] inclusion dependencies, since this assumption is sufficient to guarantee that CQA is in co-NP, see [13].

It is worth noting that, the algorithm that builds Π_{cqa} is evaluated in OntoDLV together with the ASP program produced by the OntoDLV rewriter. Since the rewriting process suitably replaces OntoDLP atoms by standard ASP atoms [4], without loss of generality we adopt in the following the standard ASP notation for atoms. Given a query Q, and a set of constraints A, Π_{cqa} is built as follows:

1- for each constraints of the form (i) in A, insert the following rule into Π_{cqa}: $\overline{a}_1(t_1) \vee \cdots \vee \overline{a}_n(t_n) :- a_1(t_1), \cdots, a_n(t_n), \sigma(t_1, \ldots, t_n)$.
2- for each atom $a(t)$ occurring in some axiom of A, insert into Π_{cqa} a rule: $a^*(t) :- a(t), not\ \overline{a}$.
3- for all constraints of the form (ii) in A, insert the following rules in Π_{cqa}: $\overline{\overline{a_1}}(t) :- a_1^*(t_1), not\ a_2^*(t)$.
4- for each $a(t)$ occurring in some axiom of A insert into Π_{cqa} the following rules: $a^r(t) :- a^*, not\ \overline{a}, not\ \overline{\overline{a}}$.

[8] Axioms of type (ii) can model inclusion dependencies under the assumption of complete sources, where facts that are not in the ontology are considered to be false.

[9] Informally, a set of inclusion dependencies is acyclic if no attribute of a relation R transitively depends (w.r.t. inclusion dependencies) on an attribute of the same R.

Finally, Q_{cqa} is built from Q by replacing atoms $a(t)$ by $a^r(t)$, whenever $a(t)$ occurs in both Q and some constraint in A. The disjunctive rules (step 1) guess atoms to be cancelled (step 2) for satisfying denial constraints, and rules generated by step 3, remove atoms violating also referential integrity constraints; eventually, step 4 builds repaired relations. Note that the minimality of answer sets guarantees that deletions are minimized.

As an example consider two relations *m(code)*, and *e(code,name)*. Suppose that the axioms are $:- e(X,Y), e(X,Z), Y <> Z.$, $:- e(X,Y), e(Z,Y), X <> Z.$ and $:- m(X), not\ code(X)$, where $code(X) :- e(X,Y).$ requiring that both *code* and *name* are keys for e and $m[code] \subseteq e[code]$. Suppose now that, the following facts are true $e(1,a), e(2,b), e(2,a), m(1), m(2)$; it can be easily verified that all the axioms are violated and $m(2)$ is consistently true. The program obtained by rewriting the constraints is:

$\overline{e}(X,Y) \vee \overline{e}(X,Z) :- e(X,Y), e(X,Z), Y <> Z.$
$\overline{e}(X,Y) \vee \overline{e}(Z,Y) :- e(X,Y), e(Z,Y), X <> Z.$
$e^*(X,Y) :- e(X,Y), not\ \overline{e}(X,Y).$ $\quad m^*(X) :- m(X), not\ \overline{m}(X).$
$code^*(X) :- code(X), not\ \overline{code}(X).$ $\quad \overline{\overline{m}}(M) :- m^*(M), not\ code^*(M).$
$m^r(X) :- m^*(X), not\ \overline{m}(X), not\ \overline{\overline{m}}(X).$
$e^r(X,Y) :- e^*(X,Y), not\ \overline{e}(X,Y), not\ \overline{\overline{e}}(X,Y).$

and the two answer sets of this program both contain $m^r(2)$, thus, $m(2)$? is derived to be consistently true.

6 Related Work

As a matter of fact, the problem of linking ontology to databases is not new [2]. Most of the available ontology systems and tools are able to deal with several sources of information by exploiting different ontology languages (see [24, 25]). Among them, the most closely related systems, which offer the possibility to import relational databases into ontologies, are: the Ontobroker system [26, 27], and the Neon tookit[10]. Both of them support a fragment of Flogic [28], and allows one to link relational database to Flogic ontologies. Comparing our approach with the above mentioned ones, we notice that, OntoDLV supports a rule-based language (ASP programs under the answer sets semantics) that, is strictly more expressive in the propositional case, and retains decidability in the general case (programs with variables). This allows to directly exploit the obtained ontology specification for solving complex reasoning tasks; moreover, the advanced data-integration features supported by OntoDLV, like consistent query answering, are missing in the above mentioned systems, which, instead, support also the integration of sources different from databases.

Another related system is MASTRO [20], that allows for liking a set of pre-existing data sources to ontologies specified in the description logic DL-Lite$_{\mathcal{A}}$. In this approach, a very similar solution for creating object identifiers form database values is used and, query answering on the obtained ontology is very efficient/scalable; it can be performed in LogSpace in the size of the original database [29, 20]. Indeed, satisfiability checking and query answering in DL-Lite$_{\mathcal{A}}$ can be carried out

[10] http://www.neon-toolkit.org/

by exploiting unfolding [20], where queries on the ontology are replaced by equivalent SQL specifications on the databases containing the A-Box. This makes the solution proposed in [20] very effective when dealing with large databases, and complexity-wise cheaper than our approach. However, the language of OntoDLV is rule-based and, thus, allows for specifying more complex queries. Indeed, OntoDLP combines (in a decidable framework) ontologies with recursive rules and non-monotonic negation. Importantly, when the specified logic program is stratified and non-disjunctive, queries are unfolded, and computation is performed in mass-memory by exploiting DLV^{DB} [22]. Note that, since the language of DLV^{DB} [22] is strictly more expressive than SQL (thanks to recursion and stratified negation), OntoDLV allows for the execution of more sophisticated queries w.r.t. [20].

Finally, since OntoDLP can be seen as an extension of disjunctive datalog with object-oriented constructs, our work is related also to the techniques proposed in the field of object-oriented databases for mapping relational data to object-views (see e.g. [30, 31]).

7 Conclusion and Future Work

In this paper we proposed a solution that allows one to "upgrade" one or more existing enterprice relational databases to an ontology. The result is the natural combination of the advantages of an ontology language (clean high-level view of the information and powerful reasoning capabilities) with the efficient exploitation of large already-existent databases.

This was obtained by extending the OntoDLV language and system. In particular, we implemented virtual classes and virtual relations, two new modeling constructs that allow the knowledge engineer to define the instances of an ontology by means of special logic rules, which act as a mapping from the information stored in database tables to concept instances. Moreover, in order to deal with consistency problems that may arise when data residing in different sources are combined in a unified ontological view [2], we developed in OntoDLV*consistent query answering* [2, 10–13], so that the system is able to retrieve as much consistent information as possible from the ontology.

Ongoing work concerns the analysis of performances of our system on real-life and large scale databases.

References

1. Gruber, T. R..: A Translation Approach to Portable Ontology Specifications. Knowledge Acquisition **5** (1993) 199–220"
2. Lenzerini, M.: Data integration: a theoretical perspective. In Popa, L., ed.: PODS '02: Proc. of PODS, New York, USA, ACM (2002) 233–246
3. Ricca, F., Gallucci, L., Schindlauer, R., Dell'Armi, T., Grasso, G., Leone, N.: OntoDLV: an ASP-based System for Enterprise Ontologies. JLC (2008) in print.
4. Ricca, F., Leone, N.: Disjunctive Logic Programming with types and objects: The DLV$^+$ System. Journal of Applied Logics **5** (2007) 545–573
5. Dell'Armi, T., Gallucci, L., Leone, N., Ricca, F., Schindlauer, R.: OntoDLV: an ASP-based System for Enterprise Ontologies. In: Proceedings ASP07. (2007)

6. Gelfond, M., Lifschitz, V.: Classical Negation in Logic Programs and Disjunctive Databases. New Generation Computing **9** (1991) 365–385
7. Gelfond, M., Leone, N.: Logic Programming and Knowledge Representation – the A-Prolog perspective . Artificial Intelligence **138** (2002) 3–38
8. Minker, J.: Overview of Disjunctive Logic Programming. Annals of Mathematics and Artificial Intelligence **12** (1994) 1–24
9. Eiter, T., Gottlob, G., Mannila, H.: Disjunctive Datalog. ACM Transactions on Database Systems **22** (1997) 364–418
10. Arenas, M., Bertossi, L.E., Chomicki, J.: Consistent Query Answers in Inconsistent Databases. In Proceedings of PODS '99), ACM Press (1999) 68–79
11. Lembo, D., Lenzerini, M., Rosati, R.: Source Inconsistency and Incompleteness in Data Integration. In: Proc. of (KRDB-02), Toulouse France, CEUR Vol-54 (2002)
12. Bertossi, L.E., Hunter, A., Schaub, T., eds.: Inconsistency Tolerance. Volume 3300 of Lecture Notes in Computer Science. Springer (2005)
13. Chomicki, J., Marcinkowski, J.: Minimal-change integrity maintenance using tuple deletions. Information and Computation **197** (2005) 90–121
14. Leone, N., Pfeifer, G., Faber, W., Eiter, T., Gottlob, G., Perri, S., Scarcello, F.: The DLV System for Knowledge Representation and Reasoning. ACM Transactions on Computational Logic **7** (2006) 499–562
15. Apt, K.R., Blair, H.A., Walker, A.: Towards a Theory of Declarative Knowledge. Morgan Kaufmann Publishers, Inc., Washington DC (1988) 89–148
16. Smith, M.K., Welty, C., McGuinness, D.L.: OWL web ontology language guide. W3C Candidate Recommendation (2003) `http://www.w3.org/TR/owl-guide/`.
17. Chen, P.P.: The Entity-Relationship Model - Toward a Unified View of Data. ACM Transactions on Database Systems **1** (1976) 9–36
18. Markowitz, V.M., Makowsky, J.A.: Identifying Extended Entity-Relationship Object Structures in Relational Schemas. IEEE Trans. Softw. Eng. **16** (1990) 777–790
19. Hull, R.: A survey of theoretical research on typed complex database objects. **15** (1987) 193–261
20. Poggi, A., Lembo, D., Calvanese, D., Giacomo, G.D., Lenzerini, M., Rosati., R.: Linking Ontologies to Data. Journal of Data Semantics (2008) 133–173
21. Gallucci, L., Ricca, F.: Visual Querying and Application Programming Interface for an ASP-based Ontology Language. In Proc. of SEA'07, AZ, USA, (2007). 56–70
22. Giorgio, T., Leone, N., Vincenzino, L., Panetta, C.: Experimenting with recursive queries in database and logic programming systems. TPLP **7** (2007) 1–37
23. Abiteboul, S., Hull, R., Vianu, V.: Foundations of Databases. Addison-Wesley (1995)
24. Duineveld, A., Stoter, R., Weiden, M., Kenepa, B., Benjamins, V.: Wonder Tools? A Comparative Study of Ontological Engineering Tools. JHCS **1** (2000) 1111–1133
25. Kalfoglou, Y., Schorlemmer, M.: Ontology mapping: the state of the art. Knowl. Eng. Rev. **18** (2003) 1–31
26. Fensel, D., Decker, S., Erdmann, M., Studer, R.: Ontobroker: How to make the www intelligent. In: In Proc. of (KAW98). (1998) 9–7
27. Sure, Y., Angele, J., Staab, S.: OntoEdit: Multifaceted Inferencing for Ontology Engineering. Journal of Data Semantics **1** (2003) 128–152
28. Kifer, M., Lausen, G., Wu, J.: Logical foundations of object-oriented and frame-based languages. Journal of the ACM **42** (1995) 741–843
29. Calvanese, D., Giacomo, G., Lembo, D., Lenzerini, M., Rosati, R.: Tractable Reasoning and Efficient Query Answering in Description Logics: The DL-Lite Family. Journal of Automated Reasoning **39** (2007) 385–429
30. Bancilhon F., Delobel C., Kanellakis P. C.: Building an Object-oriented Database System: The Story of O2. Morgan Kaufmann (1992)
31. Abiteboul S., Bonner A.: Objects and Views. ACM SIGMOD (1991) 238–247
32. ODMG: Object Data Management Group: http://www.odbms.org/.

A Sound and Complete Algorithm for Simple Conceptual Logic Programs*

Cristina Feier and Stijn Heymans

Knowledge-Based Systems Group, Institute of Information Systems
Vienna University of Technology
Favoritenstrasse 9-11, A-1040 Vienna, Austria
{feier,heymans}@kr.tuwien.ac.at

Abstract. Open Answer Set Programming (OASP) is a knowledge representation paradigm that allows for a tight integration of Logic Programming rules and Description Logic ontologies. Although several decidable fragments of OASP exist, no reasoning procedures for such expressive fragments were identified so far. We provide an algorithm that checks satisfiability in NEXPTIME for the fragment of EXPTIME-complete *simple conceptual logic programs*.

1 Introduction

Integrating Description Logics (DLs) with rules for the Semantic Web has received considerable attention over the past years with approaches such as *Description Logic Programs* [10], *DL-safe rules* [16], $\mathcal{DL}$+*log* [17], *dl-programs* [5], and Open Answer Set Programming (OASP) [13]. OASP combines attractive features from both the DL and the Logic Programming (LP) world: an open domain semantics from the DL side allows for stating generic knowledge, without mentioning actual constants, and a rule-based syntax from the LP side supports nonmonotonic reasoning via *negation as failure*.

Decidable fragments for OASP satisfiability checking were identified as syntactically restricted programs, that are still expressive enough for integrating rule- and ontology-based knowledge, see, e.g., *Conceptual Logic Programs* [12] or *g-hybrid knowledge bases* [11]. A shortcoming of those decidable fragments of OASP is the lack of effective reasoning procedures. In this paper, we take a first step in mending this by providing a sound and complete algorithm for satisfiability checking in a particular fragment of Conceptual Logic Programs.

The major contributions of the paper can be summarized as follows:

– We identify a fragment of Conceptual Logic Programs (CoLPs), called *simple CoLPs*, that disallow for inverse predicates and inequality compared to CoLPs, but are expressive enough to simulate the DL $\mathcal{SH}$. We show that

* This work is partially supported by the Austrian Science Fund (FWF) under the projects *Distributed Open Answer Set Programming (FWF P20305)* and *Reasoning in Hybrid Knowledge Bases (FWF P20840)*.

satisfiability checking w.r.t. simple CoLPs is EXPTIME-complete (i.e., it has the same complexity as CoLPs).
– We define a nondeterministic algorithm for deciding satisfiability, inspired by tableaux-based methods from DLs, that constructs a finite representation of an open answer set. We show that this algorithm is terminating, sound, complete, and runs in NEXPTIME.

The algorithm is non-trivial from two perspectives: both the minimal model semantics of OASP, compared to the model semantics of DLs, as well as the open domain assumption, compared to the closed domain assumption of ASP, pose specific challenges in constructing a finite representation that corresponds to an open answer set. Detailed proofs and an extended example can be found in [6].

2 Preliminaries

We recall the open answer set semantics from [13]. *Constants* $a, b, c, \ldots$, *variables* $x, y, \ldots$, *terms* $s, t, \ldots$, and *atoms* $p(t_1, \ldots, t_n)$ are defined as usual. A *literal* is an atom $p(t_1, \ldots, t_n)$ or a *naf-atom* $not\ p(t_1, \ldots, t_n)$. For a set α of literals or (possibly negated) predicates, $\alpha^+ = \{l \mid l \in \alpha, l \text{ an atom or a predicate}\}$ and $\alpha^- = \{l \mid not\ l \in \alpha, l \text{ an atom or a predicate}\}$. For a set X of atoms, $not\ X = \{not\ l \mid l \in X\}$. For a set of (possibly negated) predicates α, we will often write $\alpha(x)$ for $\{a(x) \mid a \in \alpha\}$ and $\alpha(x, y)$ for $\{a(x, y) \mid a \in \alpha\}$.

A *program* is a countable set of rules $\alpha \leftarrow \beta$, where α and β are finite sets of literals. The set α is the *head* of the rule and represents a disjunction, while β is called the *body* and represents a conjunction. If $\alpha = \emptyset$, the rule is called a *constraint*. *Free rules* are rules $q(x_1, \ldots, x_n) \vee not\ q(x_1, \ldots, x_n) \leftarrow$ for variables $x_1, \ldots, x_n$; they enable a choice for the inclusion of atoms. We call a predicate q *free* in a program if there is a free rule $q(x_1, \ldots, x_n) \vee not\ q(x_1, \ldots, x_n) \leftarrow$ in the program. Atoms, literals, rules, and programs that do not contain variables are *ground*. For a rule or a program X, let $cts(X)$ be the constants in X, $vars(X)$ its variables, and $preds(X)$ its predicates with $upreds(X)$ the unary and $bpreds(X)$ the binary predicates. A *universe* U for a program P is a non-empty countable superset of the constants in P: $cts(P) \subseteq U$. We call P_U the ground program obtained from P by substituting every variable in P by every possible constant in U. Let $\mathcal{B}_P$ ($\mathcal{L}_P$) be the set of atoms (literals) that can be formed from a ground program P.

An *interpretation* I of a ground P is any subset of $\mathcal{B}_P$. We write $I \models p(t_1, \ldots, t_n)$ if $p(t_1, \ldots, t_n) \in I$ and $I \models not\ p(t_1, \ldots, t_n)$ if $I \not\models p(t_1, \ldots, t_n)$. For a set of ground literals X, $I \models X$ if $I \models l$ for every $l \in X$. A ground rule $r : \alpha \leftarrow \beta$ is *satisfied* w.r.t. I, denoted $I \models r$, if $I \models l$ for some $l \in \alpha$ whenever $I \models \beta$. A ground constraint $\leftarrow \beta$ is satisfied w.r.t. I if $I \not\models \beta$. For a ground program P without *not*, an interpretation I of P is a *model* of P if I satisfies every rule in P; it is an *answer set* of P if it is a subset minimal model of P. For ground programs P containing *not*, the *GL-reduct* [7] w.r.t. I is defined as

P^I, where P^I contains $\alpha^+ \leftarrow \beta^+$ for $\alpha \leftarrow \beta$ in P, $I \models not\ \beta^-$ and $I \models \alpha^-$. I is an *answer set* of a ground P if I is an answer set of P^I.

In the following, a program is assumed to be a finite set of rules; infinite programs only appear as byproducts of grounding a finite program with an infinite universe. An *open interpretation* of a program P is a pair (U, M) where U is a universe for P and M is an interpretation of P_U. An *open answer set* of P is an open interpretation (U, M) of P with M an answer set of P_U. An n-ary predicate p in P is *satisfiable* if there is an open answer set (U, M) of P and a $(x_1, \ldots, x_n) \in U^n$ such that $p(x_1, \ldots, x_n) \in M$.

We introduce some notations for trees as in [19]. For an $x \in \mathbb{N}_0^*$ [1] we denote the concatenation of a number $c \in \mathbb{N}_0$ to x as $x \cdot c$, or, abbreviated, as xc. Formally, a *(finite) tree* T is a (finite) subset of $\mathbb{N}_0^*$ such that if $x \cdot c \in T$ for $x \in \mathbb{N}_0^*$ and $c \in \mathbb{N}_0$, then $x \in T$. Elements of T are called *nodes* and the empty word ε is the *root* of T. For a node $x \in T$ we call $succ_T(x) = \{x \cdot c \in T \mid c \in \mathbb{N}_0\}$, *successors* of x. The *arity* of a tree is the maximum amount of successors any node has in the tree. The set $A_T = \{(x, y) \mid x, y \in T, \exists c \in \mathbb{N}_0 : y = x \cdot c\}$ denotes the set of edges of a tree T. We define a partial order $\leq$ on a tree T such that for $x, y \in T$, $x \leq y$ iff x is a prefix of y. As usual, $x < y$ if $x \leq y$ and $y \not\leq x$. A (finite) *path* P in a tree T is a prefix-closed subset of T such that $\forall x \neq y \in P : |x| \neq |y|$. We call $path_T(x, y)$ a finite path in T with x the smallest element of the path w.r.t. the order relation $<$ and y the greatest element. The *length* of a finite path is the number of elements of the path. Infinite paths have no greatest element w.r.t. $<$. A branch B in a tree T is a maximal path (there is no path which contains it) which contains the root of T.

For programs containing only unary and binary predicates it makes sense to define a *tree model property*: for a program P containing only unary and binary predicates, if a unary predicate $p \in preds(P)$ is satisfiable w.r.t. P then p is tree satisfiable w.r.t. P. A predicate p is *tree satisfiable* w.r.t. P if there exists

- an open answer set (U, M) of P such that U is a tree of bounded arity, and
- a labeling function $t : U \to 2^{preds(P)}$ such that
  - $p \in t(\varepsilon)$ and $t(\varepsilon)$ does not contain binary predicates, and
  - $z \cdot i \in U$, $i > 0$, iff there is some $f(z, z \cdot i) \in M$, and
  - for $y \in U$, $q \in upreds(P)$, $f \in bpreds(P)$,
    - $q(y) \in M$ iff $q \in t(y)$, and
    - $f(x, y) \in M$ iff $y = x \cdot i \wedge f \in t(y)$.

We call such a (U, M) a *tree model* for p w.r.t. P.

3 Simple Conceptual Logic Programs

In [12], we defined *Conceptual Logic Programs (CoLPs)*, a syntactical fragment of logic programs for which satisfiability checking under the open answer set semantics is decidable. We restrict this fragment by disallowing the occurrence of inequalities and inverse predicates, resulting in *simple conceptual logic programs*.

[1] By $\mathbb{N}_0$ we denote the set of natural numbers excluding 0, and by $\mathbb{N}_0^*$ the set of finite sequences over $\mathbb{N}_0$.

Definition 1. *A* simple conceptual logic program (simple CoLP) *is a program with only unary and binary predicates, without constants, and such that any rule is a* free rule*, a* unary rule

$$a(x) \leftarrow \beta(x), \left(\gamma_m(x, y_m), \delta_m(y_m)\right)_{1 \leq m \leq k} \tag{1}$$

where for all m*,* $\gamma_m^+ \neq \emptyset$*, or a* binary rule

$$f(x, y) \leftarrow \beta(x), \gamma(x, y), \delta(y) \tag{2}$$

with $\gamma^+ \neq \emptyset$*.*

Intuitively, the free rules allow for a free introduction of atoms (in a first-order way) in answer sets, unary rules consist of a root atom $a(x)$ that is motivated by a syntactically tree-shaped body, and binary rules motivate a $f(x, y)$ for a x and its 'successor' y by a body that only considers atoms involving x and y.

Simple CoLPs can simulate constraints $\leftarrow \beta(x), \left(\gamma_m(x, y_m), \delta_m(y_m)\right)_{1 \leq m \leq k}$, where $\forall m : \gamma_m^+ \neq \emptyset$, i.e., constraints have a body that has the same form as a body of a unary rule. Indeed, such constraints $\leftarrow$ *body* can be replaced by simple CoLP rules of the form $constr(x) \leftarrow$ *not* $constr(x)$, *body*, for a new predicate *constr*.

As simple CoLPs are CoLPs and the latter have the tree model property [12], simple CoLPs have the tree model property as well.

Proposition 1. *Simple CoLPs have the tree model property.*

For CoLPs this tree model property was important to ensure that a tree automaton [19] could be constructed that accepts tree models in order to show decidability. The presented algorithm for simple CoLPs relies as well heavily on this tree model property.

As satisfiability checking of CoLPs is EXPTIME-complete [12], checking satisfiability of simple CoLPs is in EXPTIME.

In [12], it was shown that CoLPs are expressive enough to simulate satisfiability checking w.r.t to $\mathcal{SHIQ}$ knowledge bases, where $\mathcal{SHIQ}$ is the Description Logic (DL) extending $\mathcal{ALC}$ with transitive roles ($\mathcal{S}$), support for role hierarchies ($\mathcal{H}$), inverse roles ($\mathcal{I}$), and qualified number restrictions ($\mathcal{Q}$). For an overview of DLs, we refer the reader to [1].

Using a restriction of this simulation, one can show that satisfiability checking of $\mathcal{SH}$ concepts (i.e., $\mathcal{SHIQ}$ without inverse roles and quantified number restrictions) w.r.t. a $\mathcal{SH}$ TBox can be reduced to satisfiability checking of a unary predicate w.r.t. a simple CoLP. Intuitively, simple CoLPs cannot handle inverse roles (as they do not allow for inverse predicates) neither can they handle number restrictions (as they do not allow for inequality). As satisfiability checking of $\mathcal{ALC}$ concepts w.r.t. an $\mathcal{ALC}$ TBox (note that $\mathcal{ALC}$ is a fragment of $\mathcal{SH}$) is EXPTIME-complete ([1, Chapter 3]), we have EXPTIME-hardness for simple CoLPs as well.

Proposition 2. *Satisfiability checking w.r.t. simple CoLPs is* EXPTIME*-complete.*

4 An Algorithm for Simple Conceptual Logic Programs

In this section, we define a sound, complete, and terminating algorithm for satisfiability checking w.r.t. simple CoLPs.

For every non-free predicate q and a simple CoLP P, let P_q be the rules of P that have q as a head predicate. For a predicate p, $\pm p$ denotes p or *not* p, whereby multiple occurrences of $\pm p$ in the same context will refer to the same symbol (either p or *not* p). The negation of $\pm p$ is $\mp p$, that is, $\mp p = \mathit{not}\ p$ if $\pm p = p$ and $\mp p = p$ if $\pm p = \mathit{not}\ p$.

For a unary rule r of the form (1), we define $\mathit{degree}(r) = |\{m \mid \gamma_m \neq \emptyset\}|$. For every non-free rule $r : \alpha \leftarrow \beta \in P$, we assume that there exists an injective function $i_r : \beta \to \{0, \ldots, |\beta|\}$ which defines a total order over the literals in β and an inverse function $l_r : \{0, \ldots, |\beta|\} \to \beta$ which returns the literal with the given index in β. For a rule r which has body variables $x, y_1, \ldots, y_k$ we introduce a function $\mathit{varset}_r : \{x, y_1, \ldots, y_k, (x, y_1), \ldots, (x, y_k)\} \to 2^{\{0,\ldots,|\beta|\}}$ which for every variable or pair of variables which appears in at least one literal in a rule returns the set of indices of the literals formed with the corresponding variable(s).

The basic data structure for our algorithm is a *completion structure*.

Definition 2 (completion structure). *A completion structure for a simple CoLP P is a tuple $\langle T, G, \textsc{ct}, \textsc{st}, \textsc{rl}, \textsc{sg}, \textsc{nj}_{\textsc{u}}, \textsc{nj}_{\textsc{b}}\rangle$, where T is a tree which together with the labeling functions* ct, st, rl, sg, $\textsc{nj}_{\textsc{u}}$, *and* $\textsc{nj}_{\textsc{b}}$, *represents a tentative tree model and $G = \langle V, E\rangle$ is a directed graph with nodes $V \subseteq \mathcal{B}_{P_T}$ and edges $E \subseteq \mathcal{B}_{P_T} \times \mathcal{B}_{P_T}$ which is keeps track of dependencies between elements of the constructed model. The labeling functions are defined as following:*

- *The* content *function* $\textsc{ct} : T \cup A_T \to 2^{preds(P) \cup not\ (preds(P))}$ *maps a node of the tree to a set of (possibly negated) unary predicates and an edge of the tree to a set of (possibly negated) binary predicates such that* $\textsc{ct}(x) \subseteq upreds(P) \cup not(upreds(P))$ *if* $x \in T$, *and* $\textsc{ct}(x) \subseteq bpreds(P) \cup not(bpreds(P))$ *if* $x \in A_T$.
- *The* status *function* $\textsc{st} : \{(x, \pm q) \mid \pm q \in \textsc{ct}(x), x \in T \cup A_T\} \to \{exp, unexp\}$ *attaches to every (possibly negated) predicate which appears in the content of a node/edge x a status value which indicates whether the predicate has already been expanded in that node/edge.*
- *The* rule *function* $\textsc{rl} : \{(x, q) \mid x \in T \cup A_T, q \in \textsc{ct}(x)\} \to P$ *associates with every node/edge x of T and every positive predicate $q \in \textsc{ct}(x)$ a rule which has q as a head predicate:* $\textsc{rl}(x, q) \in P_q$.
- *The* segment *function* $\textsc{sg} : \{(x, q, r) \mid x \in T, \mathit{not}\ q \in \textsc{ct}(x), r \in P_q\} \to \mathbb{N}$ *indicates which part of r justifies having not q in* $\textsc{ct}(x)$.
- *The* negative justification for unary predicates *function* $\textsc{nj}_{\textsc{u}} : \{(x, q, r) \mid x \in T, \mathit{not}\ q \in \textsc{ct}(x), r \in P_q\} \to 2^{\mathbb{N} \times T}$ *indicates by means of tuples $(n, z) \in \mathbb{N} \times T$ which literal $l_r(n)$ from r is used to justify not q in* $\textsc{ct}(x)$ *in a node $z \in T$, or edge $(x, z) \in A_T$.*
- *The* negative justification for binary predicates *function* $\textsc{nj}_{\textsc{b}} : \{(x, q, r) \mid x \in A_T, \mathit{not}\ q \in \textsc{ct}(x), r \in P_q\} \to \mathbb{N}$ *gives the index of the literal from r that is used to justify not $q \in \textsc{ct}(x)$.*

An initial completion structure *for checking the satisfiability of a unary predicate p w.r.t. a simple CoLP P is a completion structure with $T = \{\varepsilon\}$, $V = \{p(\varepsilon)\}$, $E = \emptyset$, and $\textsc{ct}(\varepsilon) = \{p\}$, $\textsc{st}(\varepsilon, p) = unexp$, and the other labeling functions undefined for every input.*

We clarify the definition of a completion structure by means of an example. Take the program P :

$$\begin{array}{ll} r_1 : & f(x,y) \vee not\ f(x,y) \leftarrow \\ r_2 : & \qquad a(x) \leftarrow f(x,y_1), a(y_1), f(x,y_2) \\ r_3 : & \qquad b(x) \leftarrow not\ a(x) \end{array}$$

A possible completion structure for this program P is as follows. Take a tree $T = \{\varepsilon, \varepsilon 1\}$, i.e., a tree with root ε and successor $\varepsilon 1$, and take $\textsc{ct}(\varepsilon) = \{b, not\ a\}$, $\textsc{ct}(\varepsilon, \varepsilon 1) = \{f\}$, and $\textsc{ct}(\varepsilon 1) = \{not\ a, b\}$. Intuitively, we lay out the structure of our tree model.

We take $\textsc{rl}(\varepsilon, b) = r_3$ indicating that r_3 is responsible for motivating the occurrence of b in ε, set $\textsc{st}(\varepsilon, b) = exp$, and keep the status undefined for all other nodes and edges in T.

In general, justifying a negative unary literal $not\ q \in \textsc{ct}(x)$ (or in other words, the absence of $q(x)$ in the corresponding open interpretation) implies that every rule which defines q has to be refuted (otherwise q would have to be present), thus at least one body literal from every rule in P_q has to be refuted. A certain rule $r \in P_q$ can either be locally refuted (via a literal which can be formed using x and some $\pm a \in \textsc{ct}(x)$) or it has to be refuted in every successor of x. In the latter case, if x has more than one successor, it can be shown that the same segment of the rule has to be refuted in all the successors, whereby a segment of a rule is one of $\{\beta, (\gamma_m \cup \delta_m)_{1 \leq m \leq k}\}$ for unary rules (1). In the example, in order to have $not\ a \in \textsc{ct}(\varepsilon)$, we need that for all successors y_1, y_2, either $f \in \textsc{ct}(\varepsilon, y_1), a \in \textsc{ct}(y_1)$ does not hold, or $f \in \textsc{ct}(\varepsilon, y_2))$ does not hold; as y_1 is not appearing in the second segment (and vice versa for y_2), either for all successors y, $f \in \textsc{ct}(\varepsilon, y), a \in \textsc{ct}(y)$ does not hold, or for all successors y, $f \in \textsc{ct}(\varepsilon, y)$ does not hold, such that in our case $\textsc{sg}(x, a, r_2) = 1$ (the segment $f(x, y_1), a(y_1)$): the function $\textsc{sg}$ picks up such a segment to be refuted, where segments are referred to by the numbers 0 for β, and m for $\gamma_m \cup \delta_m$, $1 \leq m \leq k$.

After picking a segment to refute a negative unary predicate, we need means to indicate which literal in the segment, per successor, can be used to justify this negative unary predicate. This can be per successor a different literal from the segment such that $\textsc{nj}_{\textsc{u}}(x, q, r)$ is a set of tuples (n, z) where z is the particular successor (or x itself in case the negative unary predicate can be justified locally) and n the position of the literal in the rule r. In the example, $\textsc{nj}_{\textsc{u}}(x, a, r_2) = \{(1, \varepsilon 1)\}$, i.e., the literal $a(y_1)$ as $not\ a \in \textsc{ct}(\varepsilon 1)$. Note that if $z = x$ the set $\textsc{nj}_{\textsc{u}}(x, q, r)$ would be a singleton set as no successors are needed to justify $not\ q$.

Rules that can deduce negated binary predicates are always *local* in the sense that to justify a $not\ q \in \textsc{ct}(x)$ for $x \in A_T$ one only needs to consider x.

In the following, we will show how to expand the initial completion structure in order to prove satisfiability of a predicate, how to determine when no more

expansion is needed (*blocking*), and under what circumstances a *clash* occurs. In particular, *expansion rules* will expand an initial completion structure to a complete clash-free structure that corresponds to a finite representation of an open answer set; *applicability rules* state the necessary conditions such that those expansion rules can be applied.

4.1 Expansion Rules

The expansion rules will need to update the completion structure whenever in the process of justifying a literal l in the current model a new literal $\pm p(z)$ has to be considered. This means that $\pm p$ has to be inserted in the content of z in case it is not already there and marked as unexpanded, and in case $\pm p(z)$ is an atom, it has to be ensured that it is a node in G and furthermore, in case l is also an atom, a new arc from l to $\pm p(z)$ should be created to capture the dependencies between the two elements of the model. More formally:

- if $\pm p \notin \textsc{ct}(z)$, then $\textsc{ct}(z) = \textsc{ct}(z) \cup \{\pm p\}$ and $\textsc{st}(z, \pm p) = \mathit{unexp}$,
- if $\pm p = p$ and $\pm p(z) \notin V$, then $V = V \cup \{\pm p(x)\}$,
- if $l \in \mathcal{B}_{P_T}$ and $\pm p = p$, then $E = E \cup \{(l, \pm p(z))\}$.

As a shorthand, we denote this sequence of operations as $\mathit{update}(l, \pm p, z)$; more general, $\mathit{update}(l, \beta, z)$ for a set of (possibly negated) predicates β, denotes $\forall \pm a \in \beta, \mathit{update}(l, \pm a, z)$.

In the following, let $x \in T$ and $(x, y) \in A_T$ be the node, respectively edge, under consideration.

(i) Expand unary positive. For a unary positive predicate (non-free) $p \in \textsc{ct}(x)$ such that $\textsc{st}(x, p) = \mathit{unexp}$,

- nondeterministically choose a rule $r \in P_p$ of the form (1) that will motivate this predicate: set $\textsc{rl}(x, p) = r$,
- for the β in the body of this r, $\mathit{update}(p(x), \beta, x)$,
- for each $\gamma_m, 1 \leq m \leq k$, from r, nondeterministically choose a $y \in \mathit{succ}_T(x)$ or let $y = x \cdot s$, where $s \in \mathbb{N}_0^*$ s.t. $x \cdot s \notin \mathit{succ}_T(x)$ already. In the latter case, add y as a new successor of x in T: $T = T \cup \{y\}$. Take a new constant $c \in C$ s.t. $\forall z \in T : c \notin t(z)$ and update the label t for the newly created node: $t(y) = c$[2]. Next, $\mathit{update}(p(x), \gamma_m, (x, y))$ and $\mathit{update}(p(x), \delta_m, y)$.
- set $\textsc{st}(x, p) = \mathit{exp}$.

(ii) Expand unary negative. For a unary negative predicate (non-free) $\mathit{not}\ p \in \textsc{ct}(x)$ and either

[2] These constants keep track of the names of nodes in the tree, and will be useful in constructing the corresponding open answer set in the proofs of soundness and completeness; they have no role in the algorithm in itself

1. $\text{ST}(x, not\ p) = unexp$, then for every rule $r \in P_p$ of the form (1) nondeterministically choose a segment $m, 0 \leq m \leq k$: $\text{SG}(x, p, r) = m$.
   - If $m = 0$, choose a $\pm a \in \beta$, and $update(not\ p(x), \mp a, x)$, $\text{NJ}_\text{U}(x, p, r) = \{(i_r(\pm a(X)), x)\}$.
   - If $m > 0$, for every $y \in succ_T(x)$, (†) choose a $\pm a_y \in \gamma_m \cup \delta_m$, and set $\text{NJ}_\text{U}(x, p, r) = \{(i_r(\pm a_y(X, Y_m)), y) \mid \pm a_y \in \gamma_m\} \cup \{(i_r(\pm a_y(Y_m)), y) \mid \pm a_y \in \delta_m\}$. Next, $update(not\ p(x), \mp a_y, (x, y))$ if $\pm a_y \in \gamma_m$, and $update(not\ p(x), \mp a_y, y)$ if $\pm a_y \in \delta_m$.

   After every rule has been processed set $\text{ST}(x, not\ p) = exp$.
2. $\text{ST}(x, not\ p) = exp$ and for some $r \in P_p$, $\text{SG}(x, p, r) \neq 0$, and $\text{NJ}_\text{U}(x, p, r) = S$ with $|S| < |succ_T(x)|$, i.e., *not* p has already been expanded, but for some rule r it did not receive a local justification (at x), and meanwhile new successors of x have been introduced. Thus, one has to justify *not* p in the new successors as well.

   For every $r \in P_p$ of the form (1) such that $\text{SG}(x, p, r) = m \neq 0$ and for every $y \in succ_T(x)$ which has not been yet considered previously, repeat the operations in (†) as above.

(iii) Expand binary positive. For a binary positive predicate symbol (non-free) p in $\text{CT}(x, y)$ such that $\text{ST}((x, y), p) = unexp$: nondeterministically choose a rule $r \in P_p$ of the form (2) that motivates p by setting $\text{RL}((x, y), p) = r$, and $update(p(x, y), \beta, x)$, $update(p(x, y), \gamma, (x, y))$, and $update(p(x, y), \delta, y)$. Finally, set $\text{ST}((x, y), p) = exp$.

(iv) Expand binary negative. For a binary negative predicate symbol (non-free) *not* p in $\text{CT}(x, y)$ such that $\text{ST}((x, y), not\ p) = unexp$, nondeterministically choose for every rule $r \in P_p$ of the form (2) an s from $varset_r(X)$, $varset_r(X, Y)$ or $varset_r(Y)$ and let $\text{NJ}_\text{B}((x, y), p, r) = s$.

- If $s \in varset(X)$ and $\pm a(X) = l_r(s)$, $update(not\ p(x, y), \mp a, x)$,
- If $s \in varset(X, Y)$ and $\pm f(X, Y) = l_r(s)$, $update(not\ p(x, y), \mp f, (x, y))$,
- If $s \in varset(Y)$ and $\pm a(Y) = l_r(s)$, $update(not\ p(x, y), \mp a, y))$.

Finally, set $\text{ST}((x, y), not\ p) = exp$.

(v) Choose a unary predicate. There is an $x \in T$ for which none of $\pm a \in \text{CT}(x)$ can be expanded with rules (i-ii), and for all $(x, y) \in A_T$, none of $\pm f \in \text{CT}(x, y)$ can be expanded with rules (iii-iv), and there is a $p \in upreds(P)$ such that $p \notin \text{CT}(x)$ and $not\ p \notin \text{CT}(x)$. Then, add p to $\text{CT}(x)$ with $\text{ST}(x, p) = unexp$ or add *not* p to $\text{CT}(x)$ with $\text{ST}(x, not\ p) = unexp$.

(vi) Choose a binary predicate. There is an $x \in T$ for which none of $\pm a \in \text{CT}(x)$ can be expanded with rules (i-ii), and for all $(x, y) \in A_T$ none of $\pm f \in \text{CT}(x, y)$ can be expanded with rules (iii-iv), and there is a $(x, y) \in A_T$ and a $p \in bpreds(P)$ such that $p \notin \text{CT}(x, y)$ and $not\ p \notin \text{CT}(x, y)$. Then, add p to $\text{CT}(x, y)$ with $\text{ST}((x, y), p) = unexp$ or add *not* p to $\text{CT}(x, y)$ with $\text{ST}((x, y), not\ p) = unexp$.

4.2 Applicability Rules

For a simple CoLP P, a universe U for P, a graph $G = \langle V, E\rangle$ with nodes $V \in \mathcal{B}_{P_U}$ and $E \in \mathcal{B}_{P_U} \times \mathcal{B}_{P_U}$, and a set of constants $C \subseteq U$ we denote by $G(C)$ the graph obtained from G by considering only those nodes $V(C) \subseteq V$ which have an element from C as a first argument (remember that nodes are unary or binary literals) and the edges $E(C)$ which already existed between the nodes from $V(C)$ in the initial graph. Formally, $V(C) = \{\pm p(x) \mid \pm p(x) \in V \wedge x \in C\} \cup \{\pm p(x,y) \mid \pm p(x,y) \in V \wedge x \in C\}$ and $E(C) = E \cap (V(C) \times V(C))$. When U is a tree and C is a path in U, $C = path_U(x,y)$, $G(C)$ will contain those nodes from G which have as arguments nodes from C or outgoing arcs in U from nodes in C.

A second set of rules is not updating the completion structure under consideration, but restricts the use of the expansion rules:

(vii) Saturation We will call a node $x \in T$ *saturated* if

- for all $p \in upreds(P)$ we have $p \in \text{CT}(x)$ or *not* $p \in \text{CT}(x)$ and none of $\pm a \in \text{CT}(x)$ can be expanded according to the rules (i-ii) or (v),
- for all $(x,y) \in A_T$ and $p \in bpreds(P)$, $p \in \text{CT}(x,y)$ or *not* $p \in \text{CT}(x,y)$ and none of $\pm f \in \text{CT}(x,y)$ can be expanded according to the rules (iii-iv) or (vi).

We impose that no expansions (i-vi) can be performed on a node from T until its predecessor is saturated.

(viii) Blocking We call a node $x \in T$ *blocked* if

- its predecessor is saturated,
- there are two ancestors y, z such that $y < z < x$, and $\text{CT}(z) = \text{CT}(y)$, and
- $G_{y,z} = \langle V(path_T(y,z)), E(path_T(y,z)) \cup \{(a(z), a(y)) \mid a \in \text{CT}(z)\}\rangle$ is acyclic.

Note that ancestors y, z are saturated as well, by rule (vii).

Intuitively, if there is a pair of ancestor nodes that have equal content, and if by adding connections from atoms formed using the lower node in the pair to atoms formed using the higher node in the pair and the same predicate, no cycles are created in a subgraph of G which has as nodes all nodes which have as arguments nodes from the path or outgoing arcs from these nodes (the rest of G is not relevant in this context), the current node can be blocked: one can show that provided that the content of the higher node in the pair is justified, the content of the lower node in the pair can be justified also without further expansions. We call (y, z) a *blocking pair* and say that *y blocks z*; if no confusion is possible with the blocked node x, we will usually also refer to z as a blocked node and to y as the blocking node for a blocking pair (y, z). We impose that no expansions (i-vi) can be performed on a blocked node from T.

(ix) Cyclic We call a node $x \in T$ *cyclic* if

- its predecessor is saturated,
- there are two ancestors y, z such that $y < z < x$, and $\text{CT}(z) = \text{CT}(y)$, and
- $G_{y,z}$ contains a cycle.

The intuition is similar as with blocking, however, instead of being able to reuse the justification of the higher node for the lower node, the presence of a cycle in $G_{y,z}$ is indicating that we would create an infinitely positive path in G to motivate the higher node, when reusing the justification, which is, due to the minimal model semantics, not allowed. We call (y, z) a *cyclic* pair, and, when no confusion can arise, we designate z as a cyclic node as well. We impose that no expansions (i-vi) can be performed on any node in T if it contains a cyclic node.

(x) Caching We call a node $x \in T$ *cached* if

- its predecessor is saturated,
- there are two nodes y, z such that $z < x$ and $z \not\leq y$ and $y \not\leq z$ (i.e., z is not an ancestor of y nor is y an ancestor of z), and $\text{CT}(z) = \text{CT}(y)$; we call the pair (y, z) a *caching pair* and we say that y *caches* z; usually, if no confusion is possible, we will also refer to z as a cached node and to y as the caching node. Furthermore, there exists no caching pair (u, y).

We impose that no expansions can be performed on a cached node from T. Intuitively, x is not further expanded, as one can reuse the (cached) justification for y when dealing with z. The condition that there is no caching pair (u, y) ensures that we do not replace the justification of z with that one of a node y which at its turn is reusing the justification of some other node u. In particular, this precludes a situation like y caches z and z caches y.

4.3 Termination, Soundness, and Completion

We call a completion structure *contradictory*, if for some $x \in T$ and $a \in \mathit{upreds}(P)$, $\{a, \mathit{not}\ a\} \subseteq \text{CT}(x)$ or for some $(x, y) \in A_T$ and $f \in \mathit{bpreds}(P)$, $\{f, \mathit{not}\ f\} \subseteq \text{CT}(x, y)$. A *complete completion structure* for a simple CoLP P and a $p \in \mathit{upreds}(P)$, is a completion structure that results from applying the expansion rules to the initial completion structure for p and P, taking into account the applicability rules, such that no expansion rules can be further applied. Furthermore, a complete completion structure $CS = \langle T, G, \text{CT}, \text{ST}, \text{RL}, \text{SG}, \text{NJ}_\text{U}, \text{NJ}_\text{B} \rangle$ is *clash-free* if (1) CS is not contradictory, (2) t does not contain cyclic nodes, and (3) G does not contain cycles.

We show that an initial completion structure for a unary predicate p and a simple CoLP P can always be expanded to a complete completion structure (*termination*), that, if p is satisfiable w.r.t. P, there is a clash-free complete completion structure (*soundness*), and, finally, that, if there is a clash-free complete completion structure, p is satisfiable w.r.t. P (*completeness*).

Proposition 3 (termination). *Let P be a simple CoLP and $p \in upreds(P)$. Then, one can construct a finite complete completion structure by a finite number of applications of the expansion rules to the initial completion structure for p and P, taking into account the applicability rules.*

Proof Sketch. Assume one cannot construct a complete completion structure by a finite number of applications of the expansion rules, taking into account the applicability rules. Clearly, if one has a finite completion structure that is not complete, a finite application of expansion rules would complete it unless successors are introduced. However, one cannot introduce infinitely many successors: every infinite path in the tree will eventually contain two saturated nodes with equal content and thus either a blocked or a cyclic pair, such that no expansion rules can be applied to successor nodes of the blocked or cyclic node in the pair. Furthermore, the arity of the tree in the completion structure is bound by the predicates in P and the degrees of the rules. □

Proposition 4 (soundness). *Let P be a simple CoLP and $p \in upreds(P)$. If there exists a clash-free complete completion structure for p w.r.t. P, then p is satisfiable w.r.t. P.*

Proof Sketch. From a complete clash-free completion structure for p and P we can construct an open answer set of P that satisfies p by unfolding the completion structure. Intuitively, blocking pairs represent a state where the open answer set contains some infinitely repeating pattern that consists of a finite motivation for the literals in the blocking pair: the definition of a blocking pair is such that when we replace the motivation for the blocked node (i.e., the subtree below this node) by the subtree that motivates the blocking node in the pair, no infinite positive path arises. As the subtree of the blocked node is a subtree of the subtree of the blocking node, we need to repeat such a replacement infinitely. Furthermore, cached nodes represent the situation that a motivation for a node is being repeated elsewhere, such that also cached pairs will be removed by a substitution of subtrees. One can show that such a construction results in a tree model for the program. □

Proposition 5 (completeness). *Let P be a simple CoLP and $p \in upreds(P)$. If p is satisfiable w.r.t. P, then there exists a clash-free complete completion structure for p w.r.t. P.*

Proof Sketch. If p is satisfiable w.r.t. P then p is tree satisfiable w.r.t. P (Proposition 1), such that there must be a tree model (U, M) for p w.r.t. P.

One can construct a clash-free complete completion structure for p w.r.t. P, by guiding the nondeterministic application of the expansion rules by (U, M) and taking into account the constraints imposed by the saturation, blocking, caching, and clash rules.

It is worth noting that the naive application of the rules according to (U, M) does not work: the tree model might contain cyclic patterns that would result in cyclic nodes in the completion structure. However, such patterns cannot occur

infinitely (this would contradict the minimality of an open answer set), such that we can choose those expansion rules that bypass the cyclicity and immediately choose the finite motivation for a certain node. □

4.4 Complexity Results

Let $CS = \langle T, G, \text{CT}, \text{ST}, \text{RL}, \text{SG}, \text{NJ}_\text{U}, \text{NJ}_\text{B} \rangle$ be a completion structure and CS' the completion structure constructed from CS by removing from T all subtrees with roots y where (x, y) is some blocked, cyclic, or caching pair. The size of each of these subtrees is at most $k + 1$, where k is bound by the amount n of unary predicates q in P and the degrees of the rules P_q. Moreover, there are at most mk such subtrees, where m is the amount of nodes in CS'.

Assume CS' has more than 2^n nodes, then there must be two nodes $x \neq y$ such that $\text{CT}(x) = \text{CT}(y)$. If $x < y$ or $y < x$, either (x, y) or (y, x) is a blocked or cyclic pair, which contradicts the construction of CS'. If $x \not< y$ and $y \not< x$, (x, y) or (y, x) is a caching pair, again a contradiction. Thus, CS' contains at most 2^n nodes, so $m \leq 2^n$. Since CS' resulted from CS by removing at most mk subtrees of maximal size $k+1$ each, the amount of nodes in CS is $m+m(k+1) \leq (k+2)2^n$, i.e., exponential in the size of P, such that the algorithm has to visit a number of nodes that is exponential in the size of P.

The graph G has as well a number of nodes that is exponential in the size of P. Since checking for cycles in a directed graph can be done in linear time, the algorithm runs in NEXPTIME, a nondeterministic level higher than the worst-case complexity characterization (Proposition 2).

Note that such an increase in complexity is expected. For example, although satisfiability checking in $\mathcal{SHIQ}$ is EXPTIME-complete, practical algorithms run in 2-NEXPTIME [18]. Thanks to caching, however, we only have an increase to NEXPTIME.

5 Related Work

Description Logic Programs [10] represent the common subset of OWL-DL ontologies and Horn logic programs (programs without negation as failure or disjunction). As such, reasoning can be reduced to normal LP reasoning.

In [16], a clever translation of $\mathcal{SHIQ}(\mathbf{D})$ ($\mathcal{SHIQ}$ with data types) combined with *DL-safe rules* (a rule is DL-safe if each variable in the rule appears in a non-DL-atom, where a DL-atom is an atom with the predicate corresponding to a DL-concept or DL-role) to disjunctive Datalog is provided. The translation relies on a translation to clauses and subsequently applying techniques from basic superposition theory.

Reasoning in $\mathcal{DL}$+*log* [17] does not use a translation to other approaches, but defines a specific algorithm based on a partial grounding of the program and a test for containment of conjunctive queries over the DL knowledge bases. Note that [17] has a *standard names assumption* as well as a *unique names assumption* - all interpretations are over some fixed, countably infinite domain,

different constants are interpreted as different elements in that domain, and constants are in one-to-one correspondence with that domain.

dl-programs [5] have a more loosely coupled take on integrating DL knowledge bases and logic programs by allowing the program to query the DL knowledge base while as well having the possibility to send (controlled) input to the DL knowledge base. Reasoning is done via a stable model computation of the logic program, interwoven with queries that are oracles to the DL part.

Description Logic Rules[14] are defined as decidable fragments of SWRL. The rules have a tree-like structure similar to the structure of simple CoLPs rules. Depending on the underlying DL, one can distinguish between $\mathcal{SROIQ}$ rules (these do not actually extend $\mathcal{SROIQ}$, they are just syntactic sugar on top of the language), $\mathcal{EL}^{++}$ rules, DLP rules, and ELP rules [15]. The latter can be seen as an extension of both $\mathcal{EL}^{++}$ rules and DLP rules, hence their name.

The algorithm presented in Section 4 can be seen as a procedure that constructs a tableau (as is common in most DL reasoning procedures), representing the possibly infinite open answer set by a finite structure. There are several DL-based approaches which adopt a minimal-style semantics. Among this are autoepistemic[4], default[2] and circumscriptive extensions of DL[3][9]. The first two extensions are restricted to reasoning with explicitly named individuals only, while [9] allows for defeats to be based on the existence of unknown individuals. A tableau-based method for reasoning with the DL $\mathcal{ALCO}$ in the circumscriptive case has been introduced in [8]. A special preference clash condition is introduced there to distinguish between minimal and non-minimal models which is based on constructing a new classical DL knowledge base and checking its satisfiability. It would be interesting to explore the connections between our algorithm and the algorithm described there, in particular between our graph-cycle based clash condition and the preference clash condition.

6 Conclusions and Outlook

We identified a decidable class of programs, simple CoLPs, and provided a non-deterministic algorithm for checking satisfiability under the open answer set semantics that runs in NEXPTIME.

The presented algorithm is the first step in reasoning under an open answer set semantics. We intend to extend the algorithm such that it can handle the inverse predicates and inequalities of CoLPs, as well as constants. The latter would enable combined reasoning with the DL $\mathcal{SHOIQ}$ (closely related to OWL-DL) and expressive rules.

References

1. F. Baader, D. Calvanese, D. L. McGuinness, D. Nardi, and P. F. Patel-Schneider, editors. *The Description Logic Handbook: Theory, Implementation, and Applications*. Cambridge University Press, 2003.

2. F. Baader and B. Hollunder. Embedding defaults into terminological representation systems. *J. of Automated Reasoning*, 14(2):149–180, 1995.
3. P. Bonatti, C. Lutz, and F. Wolter. Expressive non-monotonic description logics based on circumscription. In *Proc. of 10th Int. Conf. on Principles of Knowledge Repr. and Reasoning (KR'06)*, pages 400–410, 2006.
4. F. M. Donini, D. Nardia, and R.Rosati. Description logics of minimal knowledge and negation as failure. *ACM Transactions on Comput. Logic*, 3(2):177–225, 2002.
5. T. Eiter, G. Ianni, T. Lukasiewicz, R. Schindlauer, and H. Tompits. Combining answer set programming with description logics for the semantic web. *Artificial Intelligence*, 172(12-13):1495–1539, 2008.
6. C. Feier and S. Heymans. A sound and complete algorithm for simple conceptual logic programs. Technical Report INFSYS RESEARCH REPORT 184-08-10, KBS Group, Technical University Vienna, Austria, October 2008. http://www.kr.tuwien.ac.at/staff/heymans/priv/projects/fwf-doasp/alpsws2008-tr.pdf.
7. M. Gelfond and V. Lifschitz. The Stable Model Semantics for Logic Programming. In *Proc. of ICLP'88*, pages 1070–1080, Cambridge, Massachusetts, 1988.
8. S. Grimm and P. Hitzler. Reasoning in circumscriptive $\mathcal{ALCO}$. Technical report, FZI at University of Karlsruhe, Germany, September 2007.
9. S. Grimm and P. Hitzler. Defeasible inference with circumscriptive OWL ontologies. In *Workshop on Advancing Reasoning on the Web: Scalability and Commonsense*, 2008.
10. B. N. Grosof, I. Horrocks, R. Volz, and S. Decker. Description logic programs: combining logic programs with description logic. In *Proc. of the World Wide Web Conf.*, pages 48–57. ACM, 2003.
11. S. Heymans, J. de Bruijn, L. Predoiu, C. Feier, and D. Van Nieuwenborgh. Guarded hybrid knowledge bases. *Theory and Practice of Logic Programming*, 8(3):411–429, 2008.
12. S. Heymans, D. Van Nieuwenborgh, and D. Vermeir. Conceptual logic programs. *Annals of Mathematics and Artificial Intelligence (Special Issue on Answer Set Programming)*, 47(1–2):103–137, June 2006.
13. S. Heymans, D. Van Nieuwenborgh, and D. Vermeir. Open answer set programming with guarded programs. *ACM Transactions on Computational Logic (TOCL)*, 9(4), October 2008.
14. M. Krötzsch, S. Rudolph, and P. Hitzler. Description logic rules. In *Proc. 18th European Conf. on Artificial Intelligence(ECAI-08)*, pages 80–84. IOS Press, 2008.
15. M. Krötzsch, S. Rudolph, and P. Hitzler. ELP: Tractable rules for OWL 2. In *Proc. 7th Int. Semantic Web Conf. (ISWC-08)*, 2008.
16. B. Motik, U. Sattler, and R. Studer. Query answering for OWL-DL with rules. *Journal of Web Semantics: Science, Services and Agents on the World Wide Web*, 3(1):41–60, July 2005.
17. R. Rosati. DL+log: Tight integration of description logics and disjunctive datalog. In *Proc. of the Int. Conf. on Principles of Knowledge Representation and Reasoning (KR)*, pages 68–78, 2006.
18. S. Tobies. *Complexity Results and Practical Algorithms for Logics in Knowledge Representation*. PhD thesis, LuFG Theoretical Computer Science, RWTH-Aachen, Germany, 2001.
19. M. Y. Vardi. Reasoning about the past with two-way automata. In *Proc. 25th Int. Colloquium on Automata, Languages and Programming*, pages 628–641. Springer-Verlag, 1998.

Combining Logic Programming with Description Logics and Machine Learning for the Semantic Web

Francesca A. Lisi and Floriana Esposito

Dipartimento di Informatica, Università degli Studi di Bari
Via E. Orabona 4, 70125 Bari, Italy
{lisi, esposito}@di.uniba.it

Abstract. In this paper we consider an extension of Logic Programming that tackles the Semantic Web challenge of acquiring rules combined with ontologies. To face this bottleneck problem we propose a framework that resorts to the expressive and deductive power of $\mathcal{DL}$+log and adopts the methodological apparatus of Inductive Logic Programming.

1 Introduction

Combining rules and ontologies is a hot topic in the (Semantic) Web area as testified by the intense activity and the standardization efforts of the Rules Interchange Format working group at W3C. Yet the debate around a unified language for (Semantic) Web rules is still open. Indeed, combining rules and ontologies raises several issues in Knowledge Representation (KR) due to the many differences between the underlying logics, Clausal Logics (CLs) [17] and Description Logics (DLs) [1] respectively. Among the many recent KR proposals, $\mathcal{DL}$+log [23] is a very powerful framework that allows for the tight integration of DLs and disjunctive DATALOG with negation (DATALOG$^{\neg\vee}$) [7]. A point in favour of $\mathcal{DL}$+log is its decidability for many DLs, notably for $\mathcal{SHIQ}$ [12]. Since the design of OWL has been based on the $\mathcal{SH}$ family of very expressive DLs [11], $\mathcal{SHIQ}$+log is a good candidate for investigation in the Semantic Web context.

The upcoming standard rule language for the Semantic Web, if well-founded from the KR viewpoint, will be equipped with reasoning algorithms. In KR tradition deductive reasoning is the most widely studied. Yet, other forms of reasoning will become necessary. E.g., acquiring and maintaining Semantic Web rules is very demanding and can be automated though partially by applying Machine Learning algorithms. In this paper, we consider a decidable instantiation of $\mathcal{DL}$+log obtained by choosing $\mathcal{SHIQ}$ for the DL part and DATALOG$^{\neg}$ for the CL part, and face the problem of defining inductive reasoning mechanisms on it. To solve the problem, we propose to resort to the methodological apparatus of that form of Machine Learning known under the name of Inductive Logic Programming (ILP) [19]. We extend some known ILP techniques to $\mathcal{SHIQ} + \text{LOG}^{\neg}$ and illustrate them with examples relevant to the Semantic Web context.

The paper is organized as follows. Section 2 briefly introduces hybrid DL-CL formalisms and ILP. Section 3 introduces the KR framework of $\mathcal{DL}$+log. Section 4 defines the ILP framework for inducing $\mathcal{SHIQ}$+log$^{\neg}$ rules. Section 5 provides a comparative analysis of our proposal with related work. Section 6 concludes the paper with final remarks.

2 Background

2.1 Logic Programming and Description Logics

Description Logics (DLs) are a family of KR formalims that allow for the specification of knowledge in terms of classes (*concepts*), binary relations between classes (*roles*), and instances (*individuals*) [1]. Complex concepts can be defined from atomic concepts and roles by means of constructors (see Table 1). E.g., concept descriptions in the basic DL $\mathcal{AL}$ are formed according to only the constructors of atomic negation, concept conjunction, value restriction, and limited existential restriction. The DLs $\mathcal{ALC}$ and $\mathcal{ALN}$ are members of the $\mathcal{AL}$ family. The former extends $\mathcal{AL}$ with (arbitrary) concept negation (or complement), whereas the latter with number restriction. The DL $\mathcal{ALCNR}$ adds to the constructors inherited from $\mathcal{ALC}$ and $\mathcal{ALN}$ a further one: role intersection (see Table 1). Conversely, in the DL $\mathcal{SHIQ}$ [12] it is allowed to invert roles and to express qualified number restrictions of the form $\geq nS.C$ and $\leq nS.C$ where S is a simple role (see Table 1).

A DL knowledge base (KB) can state both is-a relations between concepts (*axioms*) and instance-of relations between individuals (resp. couples of individuals) and concepts (resp. roles) (*assertions*). Concepts and axioms form the TBox whereas individuals and assertions form the ABox. A $\mathcal{SHIQ}$ KB encompasses also a RBox which consists of axioms concerning abstract roles. The semantics of DLs is usually defined through a mapping to First Order Logic (FOL) [2]. An *interpretation* $\mathcal{I} = (\Delta^{\mathcal{I}}, \cdot^{\mathcal{I}})$ for a DL KB consists of a non-empty domain $\Delta^{\mathcal{I}}$ and a mapping function $\cdot^{\mathcal{I}}$. In particular, individuals are mapped to elements of $\Delta^{\mathcal{I}}$ such that $a^{\mathcal{I}} \neq b^{\mathcal{I}}$ if $a \neq b$ (*Unique Names Assumption* (UNA) [21]). Yet in $\mathcal{SHIQ}$ UNA does not hold by default [10]. Thus individual equality (inequality) assertions may appear in a $\mathcal{SHIQ}$ KB (see Table 1). Also the KB represents many different interpretations, i.e. all its models. This is coherent with the *Open World Assumption* (OWA) that holds in FOL semantics. The main reasoning task for a DL KB is the *consistency check* that is performed by applying decision procedures based on tableau calculus. Decidability of reasoning is crucial in DLs.

The integration of DLs and Logic Programming follows the tradition of KR research on *hybrid systems*, i.e. those systems which are constituted by two or more subsystems dealing with distinct portions of a single KB by performing specific reasoning procedures [8], and gives raise to KR systems that will be referred to as DL-CL hybrid systems in the rest of the paper. The motivation for investigating and developing such systems is to improve on *representational adequacy* and *deductive power* by preserving *decidability*. In particular, combining DLs with CLs can easily yield to undecidability if the interface between

Table 1. Syntax and semantics of DLs.

bottom (resp. top) concept	$\bot$ (resp. $\top$)	$\emptyset$ (resp. $\Delta^{\mathcal{I}}$)
atomic concept	A	$A^{\mathcal{I}} \subseteq \Delta^{\mathcal{I}}$
(abstract) role	R	$R^{\mathcal{I}} \subseteq \Delta^{\mathcal{I}} \times \Delta^{\mathcal{I}}$
(abstract) inverse role	R^-	$(R^{\mathcal{I}})^-$
(abstract) individual	a	$a^{\mathcal{I}} \in \Delta^{\mathcal{I}}$
concept negation	$\neg C$	$\Delta^{\mathcal{I}} \setminus C^{\mathcal{I}}$
concept intersection	$C_1 \sqcap C_2$	$C_1^{\mathcal{I}} \cap C_2^{\mathcal{I}}$
concept union	$C_1 \sqcup C_2$	$C_1^{\mathcal{I}} \cup C_2^{\mathcal{I}}$
value restriction	$\forall R.C$	$\{x \in \Delta^{\mathcal{I}} \mid \forall y\ (x,y) \in R^{\mathcal{I}} \rightarrow y \in C^{\mathcal{I}}\}$
existential restriction	$\exists R.C$	$\{x \in \Delta^{\mathcal{I}} \mid \exists y\ (x,y) \in R^{\mathcal{I}} \wedge y \in C^{\mathcal{I}}\}$
at least number restriction	$\geq nR$	$\{x \in \Delta^{\mathcal{I}} \mid \lvert\{y \lvert (x,y) \in R^{\mathcal{I}}\}\rvert \geq n\}$
at most number restriction	$\leq nR$	$\{x \in \Delta^{\mathcal{I}} \mid \lvert\{y \lvert (x,y) \in R^{\mathcal{I}}\}\rvert \leq n\}$
at least qualif. number restriction	$\geq nS.C$	$\{x \in \Delta^{\mathcal{I}} \mid \lvert\{y \in C^{\mathcal{I}} \lvert (x,y) \in S^{\mathcal{I}}\}\rvert \geq n\}$
at most qualif. number restriction	$\leq nS.C$	$\{x \in \Delta^{\mathcal{I}} \mid \lvert\{y \in C^{\mathcal{I}} \lvert (x,y) \in S^{\mathcal{I}}\}\rvert \leq n\}$
role intersection	$R_1 \sqcap R_2$	$R_1^{\mathcal{I}} \cap R_2^{\mathcal{I}}$
concept equivalence axiom	$C_1 \equiv C_2$	$C_1^{\mathcal{I}} = C_2^{\mathcal{I}}$
concept subsumption axiom	$C_1 \sqsubseteq C_2$	$C_1^{\mathcal{I}} \subseteq C_2^{\mathcal{I}}$
role equivalence axiom	$R \equiv S$	$R^{\mathcal{I}} = S^{\mathcal{I}}$
role inclusion axiom	$R \sqsubseteq S$	$R^{\mathcal{I}} \subseteq S^{\mathcal{I}}$
concept assertion	$a : C$	$a^{\mathcal{I}} \in C^{\mathcal{I}}$
role assertion	$\langle a, b\rangle : R$	$(a^{\mathcal{I}}, b^{\mathcal{I}}) \in R^{\mathcal{I}}$
individual equality assertion	$a \approx b$	$a^{\mathcal{I}} = b^{\mathcal{I}}$
individual inequality assertion	$a \not\approx b$	$a^{\mathcal{I}} \neq b^{\mathcal{I}}$

them is not reduced. In [14] the family CARIN of languages combining any DL and HCL is presented. Among the many important results of this study, it is proved that query answering in a logic obtained by extending $\mathcal{ALCNR}$ with non-recursive DATALOG rules, where both concepts and roles can occur in rule bodies, is decidable. Query answering is decided using *constrained SLD-resolution*, i.e. an extension of SLD-resolution with a modified version of tableau calculus. Another DL-CL hybrid system is $\mathcal{AL}$-log [6] that integrates $\mathcal{ALC}$ [26] and DATALOG [4] by constraining the variables occurring in the body of rules with $\mathcal{ALC}$ concept assertions. Constrained SLD-resolution for $\mathcal{AL}$-log is decidable and offers a complete and sound method for answering ground queries by refutation. Besides decidability, another relevant issue is *DL-safeness* of hybrid DL-CL systems [22]. A safe interaction between the DL and the CL part of an hybrid KB allows to solve the semantic mismatch between DLs and CLs due to the different inferences that can be made under OWA and CWA respectively. In this respect, $\mathcal{AL}$-log is DL-safe whereas CARIN is not.

2.2 Logic Programming and Machine Learning

The research area born at the intersection of Logic Programming and Machine Learning, more precisely Concept Learning [18], is known under the name of Inductive Logic Programming (ILP) [19]. From Logic Programming ILP has borrowed the KR framework, i.e. Horn Clausal Logic (HCL). From Concept Learning it has inherited the inferential mechanisms for induction, the most prominent of which is *generalization* characterized as search through a partially ordered space of hypotheses. According to this vision, in ILP a hypothesis is a clausal theory (i.e., a set of rules) and the induction of a single clause (rule) requires (i) structuring, (ii) searching and (iii) bounding the space of hypotheses. First we focus on (i) by clarifying the notion of *ordering* for clauses. An ordering allows for determining which one, between two clauses, is more general than the other. Actually quasi-orders are considered, therefore uncomparable pairs of clauses are admitted. One such ordering is *θ-subsumption* [20]: Given two clauses C and D, we say that C θ-subsumes D if there exists a substitution θ, such that $C\theta \subseteq D$. Given the usefulness of Background Knowledge (BK) in ILP, orders have been proposed that reckon with it, e.g. Buntine's *generalized subsumption* [3]. Generalized subsumption only applies to definite clauses and the BK should be a definite program. Once structured, the space of hypotheses can be searched (ii) by means of refinement operators. A *refinement operator* is a function which computes a set of specializations or generalizations of a clause according to whether a top-down or a bottom-up search is performed. The two kinds of refinement operator have been therefore called *downward* and *upward*, respectively. The definition of refinement operators presupposes the investigation of the properties of the various quasi-orders and is usually coupled with the specification of a declarative bias for bounding the space of clauses (iii). *Bias* concerns anything which constrains the search for theories, e.g. a *language bias* specifies syntactic constraints on the clauses in the search space.

Induction with ILP generalizes from individual instances/observations in the presence of BK, finding *valid hypotheses*. Validity depends on the underlying *setting*. At present, there exist several formalizations of induction in clausal logic that can be classified according to the following two orthogonal dimensions: the *scope of induction* (discrimination vs characterization) and the *representation of observations* (ground definite clauses vs ground unit clauses) [5]. *Discriminant induction* aims at inducing hypotheses with discriminant power as required in tasks such as classification. In classification, observations encompass both positive and negative examples. *Characteristic induction* is more suitable for finding regularities in a data set. This corresponds to learning from positive examples only. The second dimension affects the notion of *coverage*, i.e. the condition under which a hypothesis explains an observation. In *learning from entailment* (or *from implications*), hypotheses are clausal theories, observations are ground definite clauses, and a hypothesis covers an observation if the hypothesis logically entails the observation. In *learning from interpretations*, hypotheses are clausal theories, observations are Herbrand interpretations (ground unit clauses) and a hypothesis covers an observation if the observation is a model for the hypothesis.

3 Combining LP and DLs with $\mathcal{DL}$+log

The KR framework of $\mathcal{DL}$+log [23] allows for the tight integration of DLs [1] and DATALOG$^{\neg\vee}$ [7]. More precisely, it allows a DL KB to be extended with *weakly-safe* DATALOG$^{\neg\vee}$ rules. The condition of weak safeness allows to overcome the main representational limits of the approaches based on the DL-safeness condition, e.g. the possibility of expressing conjunctive queries (CQ) and unions of conjunctive queries (UCQ)[1], by keeping the integration scheme still decidable. To a certain extent, $\mathcal{DL}$+log is between $\mathcal{AL}$-log [6] and CARIN [14].

3.1 Syntax

Formulas in $\mathcal{DL}$+log are built upon three mutually disjoint predicate alphabets: an alphabet of concept names P_C, an alphabet of role names P_R, and an alphabet of DATALOG predicates P_D. We call a predicate p a *DL-predicate* if either $p \in P_C$ or $p \in P_R$. Then, we denote by $\mathcal{C}$ a countably infinite alphabet of constant names. An *atom* is an expression of the form $p(X)$, where p is a predicate of arity n and X is a n-tuple of variables and constants. If no variable symbol occurs in X, then $p(X)$ is called a *ground atom* (or *fact*). If $p \in P_C \cup P_R$, the atom is called a *DL-atom*, while if $p \in P_D$, it is called a DATALOG *atom*.

Given a description logic $\mathcal{DL}$, a $\mathcal{DL}$+log KB $\mathcal{B}$ is a pair (Σ, Π), where Σ is a $\mathcal{DL}$ KB and Π is a set of DATALOG$^{\neg\vee}$ rules, where each rule R has the form

$$p_1(\boldsymbol{X_1}) \vee \ldots \vee p_n(\boldsymbol{X_n}) \leftarrow$$
$$r_1(\boldsymbol{Y_1}), \ldots, r_m(\boldsymbol{Y_m}), s_1(\boldsymbol{Z_1}), \ldots, s_k(\boldsymbol{Z_k}), \neg u_1(\boldsymbol{W_1}), \ldots, \neg u_h(\boldsymbol{W_h})$$

with $n, m, k, h \geq 0$, each $p_i(\boldsymbol{X_i})$, $r_j(\boldsymbol{Y_j})$, $s_l(\boldsymbol{Z_l})$, $u_k(\boldsymbol{W_k})$ is an atom and:

- each p_i is either a DL-predicate or a DATALOG predicate;
- each r_j, u_k is a DATALOG predicate;
- each s_l is a DL-predicate;
- (DATALOG safeness) every variable occurring in R must appear in at least one of the atoms $r_1(\boldsymbol{Y_1}), \ldots, r_m(\boldsymbol{Y_m}), s_1(\boldsymbol{Z_1}), \ldots, s_k(\boldsymbol{Z_k})$;
- (weak safeness) every head variable of R must appear in at least one of the atoms $r_1(\boldsymbol{Y_1}), \ldots, r_m(\boldsymbol{Y_m})$.

We remark that the above notion of weak safeness allows for the presence of variables that only occur in DL-atoms in the body of R. On the other hand, the notion of DL-safeness can be expressed as follows: every variable of R must appear in at least one of the atoms $r_1(\boldsymbol{Y_1}), \ldots, r_m(\boldsymbol{Y_m})$. Therefore, DL-safeness forces every variable of R to occur also in the DATALOG atoms in the body of R, while weak safeness allows for the presence of variables that only occur in DL-atoms in the body of R. Without loss of generality, we can assume that in a $\mathcal{DL}$+log KB (Σ, Π) all constants occurring in Σ also occur in Π.

[1] A *Boolean UCQ* over a predicate alphabet P is a first-order sentence of the form $\exists \boldsymbol{X}.conj_1(\boldsymbol{X}) \vee \ldots \vee conj_n(\boldsymbol{X})$, where $\boldsymbol{X}$ is a tuple of variable symbols and each $conj_i(\boldsymbol{X})$ is a set of atoms whose predicates are in P and whose arguments are either constants or variables from $\boldsymbol{X}$. A *Boolean CQ* corresponds to a Boolean UCQ in the case when $n = 1$.

Example 1. Let us consider a $\mathcal{DL}$+log KB $\mathcal{B}$ (adapted from [23]) integrating the following DL-KB $\mathcal{Σ}$ (ontology about persons)

$[A1]$ PERSON $\sqsubseteq \exists$ FATHER$^-$.MALE
$[A2]$ MALE $\sqsubseteq$ PERSON
$[A3]$ FEMALE $\sqsubseteq$ PERSON
$[A4]$ FEMALE $\sqsubseteq \neg$MALE
MALE(Bob)
PERSON(Mary)
PERSON(Paul)
FATHER(John,Paul)

and the following DATALOG$^{\neg\vee}$ program Π (rules about students):

$[R1]$ boy(X) $\leftarrow$ enrolled(X,c1,bsc), PERSON(X), $\neg$girl(X)
$[R2]$ girl(X) $\leftarrow$ enrolled(X,c2,msc), PERSON(X)
$[R3]$ boy(X)$\vee$ girl(X) $\leftarrow$ enrolled(X,c3,phd), PERSON(X)
$[R4]$ FEMALE(X) $\leftarrow$ girl(X)
$[R5]$ MALE(X) $\leftarrow$ boy(X)
$[R6]$ man(X) $\leftarrow$ enrolled(X,c3,phd), FATHER(X,Y)
enrolled(Paul,c1,bsc)
enrolled(Mary,c1,bsc)
enrolled(Mary,c2,msc)
enrolled(Bob,c3,phd)
enrolled(John,c3,phd)

Note that the rules mix DL-literals and DATALOG-literals. Notice that the variable Y in rule $R6$ is weakly-safe but not DL-safe, since Y does not occur in any DATALOG predicate in $R6$.

3.2 Semantics

For $\mathcal{DL}$+log two semantics have been defined: a first-order logic (FOL) semantics and a nonmonotonic (NM) semantics. In particular, the latter extends the stable model semantics of DATALOG$^{\neg\vee}$ [9]. According to it, DL-predicates are still interpreted under OWA, while DATALOG predicates are interpreted under CWA. Notice that, under both semantics, entailment can be reduced to satisfiability. In a similar way, it can be seen that CQ answering can be reduced to satisfiability in $\mathcal{DL}$+log. Consequently, Rosati [23] concentrates on the satisfiability problem in $\mathcal{DL}$+log KBs. It has been shown that, when the rules are positive disjunctive, the above two semantics are equivalent with respect to the satisfiability problem. In particular, FOL-satisfiability can always be reduced (in linear time) to NM-satisfiability. Hence, the satisfiability problem under the NM semantics is in the focus of interest.

Example 2. With reference to Example 1, it can be easily verified that all NM-models for $\mathcal{B}$ satisfy the following ground atoms:

– `boy(Paul)` (since rule R1 is always applicable for {`X/Paul`} and R1 acts like a default rule, which can be read as follows: if `X` is a person enrolled in course `c1`, then `X` is a boy, unless we know for sure that `X` is a girl);
– `girl(Mary)` (since rule R2 is always applicable for {`X/Mary`});
– `boy(Bob)` (since rule R3 is always applicable for {`X/Bob`}, and, by rule R4, the conclusion `girl(Bob)` is inconsistent with Σ);
– `MALE(Paul)` (due to rule R5);
– `FEMALE(Mary)` (due to rule R4).

Notice that $\mathcal{B} \models_{NM}$`FEMALE(Mary)`, while $\Sigma \not\models_{FOL}$ `FEMALE(Mary)`. In other words, adding rules has indeed an effect on the conclusions one can draw about DL-predicates. Moreover, such an effect also holds under the FOL semantics of $\mathcal{DL}$+log-KBs, since it can be verified that $\mathcal{B} \models_{FOL}$`FEMALE(Mary)` in this case.

3.3 Reasoning

The problem statement of satisfiability for finite $\mathcal{DL}$+log KBs relies on the following problem known as the *Boolean CQ/UCQ containment problem*[2] in DLs: Given a $\mathcal{DL}$-TBox $\mathcal{T}$, a Boolean CQ Q_1 and a Boolean UCQ Q_2 over the alphabet $P_C \cup P_R$, Q_1 is contained in Q_2 with respect to $\mathcal{T}$, denoted by $\mathcal{T} \models Q_1 \subseteq Q_2$, iff, for every model $\mathcal{I}$ of $\mathcal{T}$, if Q_1 is satisfied in $\mathcal{I}$ then Q_2 is satisfied in $\mathcal{I}$. The algorithm NMSAT-$\mathcal{DL}$+log for deciding NM-satisfiability of $\mathcal{DL}$+log KBs looks for a guess (G_P, G_N) of the Boolean CQs in the DL-grounding of Π, denoted as $gr_p(\Pi)$, that is consistent with the $\mathcal{DL}$-KB Σ (Boolean CQ/UCQ containment problem) and such that the DATALOG$^{\neg\vee}$ program $\Pi(G_P, G_N)$ has a stable model. Details of how obtaining $gr_p(\Pi)$ and $\Pi(G_P, G_N)$ can be found in [23].

The decidability of reasoning in $\mathcal{DL}$+log, thus of ground query answering, depends on the decidability of the Boolean CQ/UCQ containment problem in $\mathcal{DL}$. Consequently, ground queries can be answered by applying NMSAT-$\mathcal{DL}$+log.

Theorem 1 *[23] For every description logic $\mathcal{DL}$, satisfiability of $\mathcal{DL}$+log-KBs (both under FOL semantics and under NM semantics) is decidable iff Boolean CQ/UCQ containment is decidable in $\mathcal{DL}$.*

Corollary 1. *Given a $\mathcal{DL}$+log KB (Σ, Π) and a ground atom α, $(\Sigma, \Pi) \models \alpha$ iff $(\Sigma, \Pi \cup \{\leftarrow \alpha\})$ is unsatisfiable.*

From Theorem 1 and from previous results on query answering and query containment in DLs, it follows the decidability of reasoning in several instantiations of $\mathcal{DL}$+log. Since $\mathcal{SHIQ}$ is the most expressive DL for which the Boolean CQ/UCQ containment is decidable [10], we consider $\mathcal{SHIQ}$+log$^{\neg}$ (i.e. $\mathcal{SHIQ}$ extended with weakly-safe DATALOG$^{\neg}$ rules) as the KR framework in our study of ILP for the Semantic Web.

[2] This problem was called *existential entailment* in [14].

4 Inducing $\mathcal{SHIQ}$+log$^{\neg}$ Rules with ILP

We consider the task of inducing new $\mathcal{SHIQ}$+log$^{\neg}$ rules from an already existing $\mathcal{SHIQ}$+log$^{\neg}$ KB. At this stage of work the scope of induction does not matter. Therefore the term 'observation' is to be preferred to the term 'example'. We choose to work within the setting of *learning from interpretations* which requires an observation to be represented as a set of ground unit clauses.

We assume that the data are represented as a $\mathcal{SHIQ}$+log$^{\neg}$ KB $\mathcal{B}$ where the intensional part $\mathcal{K}$ (i.e., the TBox $\mathcal{T}$ plus the set Π_R of rules) plays the role of *background knowledge* and the extensional part (i.e., the ABox $\mathcal{A}$ plus the set Π_F of facts) contributes to the definition of *observations*. Therefore ontologies may appear as input to the learning problem of interest.

Example 3. Suppose we have a $\mathcal{SHIQ}$+log$^{\neg}$ KB (adapted from [23]) consisting of the following intensional knowledge $\mathcal{K}$:

[$A1$] $\texttt{RICH} \sqcap \texttt{UNMARRIED} \sqsubseteq \exists\ \texttt{WANTS-TO-MARRY}^{-}.\top$
[$R1$] $\texttt{RICH(X)} \leftarrow \texttt{famous(X)}, \neg\ \texttt{scientist(X)}$

and the following extensional knowledge $\mathcal{F}$:

```
UNMARRIED(Mary)
UNMARRIED(Joe)
famous(Mary)
famous(Paul)
famous(Joe)
scientist(Joe)
```

that can be split into $\mathcal{F}_{\texttt{Joe}} = \{\texttt{UNMARRIED(Joe)}, \texttt{famous(Joe)}, \texttt{scientist(Joe)}\}$, $\mathcal{F}_{\texttt{Mary}} = \{\texttt{UNMARRIED(Mary)}, \texttt{famous(Mary)}\}$, and $\mathcal{F}_{\texttt{Paul}} = \{\texttt{famous(Paul)}\}$.

The language $\mathcal{L}$ of hypotheses must allow for the generation of $\mathcal{SHIQ}$+log$^{\neg}$ rules starting from three disjoint alphabets $P_C(\mathcal{L}) \subseteq P_C(\mathcal{B})$, $P_R(\mathcal{L}) \subseteq P_R(\mathcal{B})$, and $P_D(\mathcal{L}) \subseteq P_D(\mathcal{B})$. More precisely, we consider linked[3] and range-restricted[4] weakly-safe DATALOG$^{\neg}$ clauses of the form

$$p(\boldsymbol{X}) \leftarrow r_1(\boldsymbol{Y_1}), \ldots, r_m(\boldsymbol{Y_m}), s_1(\boldsymbol{Z_1}), \ldots, s_k(\boldsymbol{Z_k}), \neg u_1(\boldsymbol{W_1}), \ldots, \neg u_h(\boldsymbol{W_h})$$

where the unique literal $p(\boldsymbol{X})$ in the head represents the target predicate, denoted as c if p is a DATALOG-predicate and as C if p is a $\mathcal{SHIQ}$-predicate. In the following we provide examples for these two cases of rule learning, one aimed at inducing $c(\boldsymbol{X}) \leftarrow$ rules and the other $C(\boldsymbol{X}) \leftarrow$ rules. The former kind of rule will enrich the DATALOG part of the KB, whereas the latter will extend the DL part (i.e., the input ontology).

[3] A clause H is *linked* if each literal $l_i \in H$ is linked. A literal $l_i \in H$ is linked if at least one of its terms is linked. A term t in some literal $l_i \in H$ is linked with linking-chain of length 0, if t occurs in $head(H)$, and with linking-chain of length $d+1$, if some other term in l_i is linked with linking-chain of length d. The link-depth of a term t in l_i is the length of the shortest linking-chain of t.

[4] A clause H is *range-restricted* if each variable occurring in $head(H)$ also occur in $body(H)$.

Example 4. Suppose that the DATALOG-predicate `happy` is the target predicate and the set $P_D(\mathcal{L}^{\texttt{happy}}) \cup P_C(\mathcal{L}^{\texttt{happy}}) \cup P_R(\mathcal{L}^{\texttt{happy}}) = \{\texttt{famous/1}\} \cup \{\texttt{RICH/1}\} \cup \{\texttt{WANTS-TO-MARRY/2}, \texttt{LIKES/2}\}$ provides the building blocks for the language $\mathcal{L}^{\texttt{happy}}$. The following $\mathcal{SHIQ}^{\neg}$ rules

$H_1^{\texttt{happy}}$ `happy(X)` $\leftarrow$ `RICH(X)`
$H_2^{\texttt{happy}}$ `happy(X)` $\leftarrow$ `famous(X)`
$H_3^{\texttt{happy}}$ `happy(X)` $\leftarrow$ `famous(X)`, `WANTS-TO-MARRY(Y,X)`

belonging to $\mathcal{L}^{\texttt{happy}}$ can be considered hypotheses for the target predicate `happy`. Note that $H_3^{\texttt{happy}}$ is weakly-safe.

Example 5. Suppose now that the target predicate is the DL-predicate `LONER`. If $\mathcal{L}^{\texttt{LONER}}$ is defined over $P_D(\mathcal{L}^{\texttt{LONER}}) \cup P_C(\mathcal{L}^{\texttt{LONER}}) = \{\texttt{famous/1}, \texttt{scientist/1}\} \cup \{\texttt{UNMARRIED/1}\}$, then the following $\mathcal{SHIQ}^{\neg}$ rules

$H_1^{\texttt{LONER}}$ `LONER(X)` $\leftarrow$ `scientist(X)`
$H_2^{\texttt{LONER}}$ `LONER(X)` $\leftarrow$ `scientist(X)`, `UNMARRIED(X)`
$H_3^{\texttt{LONER}}$ `LONER(X)` $\leftarrow \neg$`famous(X)`

belong to $\mathcal{L}^{\texttt{LONER}}$ and represent hypotheses for the target predicate `LONER`.

In order to support with ILP techniques the induction of $\mathcal{SHIQ}^{\neg}$ rules, the language $\mathcal{L}$ of hypotheses needs to be equipped with a generality order $\succeq$, and a coverage relation *covers* so that $(\mathcal{L}, \succeq)$ is a search space and *covers* defines the mappings from $(\mathcal{L}, \succeq)$ to the set O of observations. The next subsections are devoted to these issues.

4.1 The hypothesis ordering

The definition of a generality order for hypotheses in $\mathcal{L}$ can disregard neither the peculiarities of $\mathcal{SHIQ}^{\neg}$ nor the methodological apparatus of ILP. One issue arises from the presence of NAF literals (i.e., negated DATALOG literals) both in the background knowledge and in the language of hypotheses. As pointed out in [25], rules in normal logic programs are syntactically regarded as Horn clauses by viewing the NAF-literal $\neg p(X)$ as an atom $not\_p(X)$ with the new predicate not_p. Then any result obtained on Horn logic programs is directly carried over to normal logic programs. Assuming one such treatment of NAF literals, we propose to adapt generalized subsumption [3] to the case of $\mathcal{SHIQ}^{\neg}$ rules. The resulting generality relation will be called $\mathcal{K}$-*subsumption*, briefly $\succeq_{\mathcal{K}}$, from now on. We provide a characterization of $\succeq_{\mathcal{K}}$ that relies on the reasoning tasks known for $\mathcal{DL}$+log and from which a test procedure can be derived.

Definition 1. *Let $H_1, H_2 \in \mathcal{L}$ be two hypotheses standardized apart, $\mathcal{K}$ a background knowledge, and σ a Skolem substitution for H_2 with respect to $\{H_1\} \cup \mathcal{K}$. We say that $H_1 \succeq_{\mathcal{K}} H_2$ iff there exists a ground substitution θ for H_1 such that (i) $head(H_1)\theta = head(H_2)\sigma$ and (ii) $\mathcal{K} \cup body(H_2)\sigma \models body(H_1)\theta$.*

Note that condition (ii) is a variant of the Boolean CQ/UCQ containment problem because $body(H_2)\sigma$ and $body(H_1)\theta$ are both Boolean CQs. The difference between (ii) and the original formulation of the problem is that $\mathcal{K}$ encompasses not only a TBox but also a set of rules. Nonetheless this variant can be reduced to the satisfiability problem for finite $\mathcal{SHIQ}$+log$^{\neg}$ KBs. Indeed the skolemization of $body(H_2)$ allows to reduce the Boolean CQ/UCQ containment problem to a CQ answering problem[5]. Due to the aforementioned link between CQ answering and satisfiability, checking (ii) can be reformulated as proving that the KB $(\mathcal{T}, \Pi_R \cup body(H_2)\sigma \cup \{\leftarrow body(H_1)\theta\})$ is unsatisfiable. Once reformulated this way, (ii) can be solved by applying the algorithm NMSAT-$\mathcal{DL}$+log.

Example 6. Let us consider the hypotheses

$H_1^{\texttt{happy}}$ `happy(A)` $\leftarrow$ `RICH(A)`
$H_2^{\texttt{happy}}$ `happy(X)` $\leftarrow$ `famous(X)`

reported in Example 4 up to variable renaming. We want to check whether $H_1^{\texttt{happy}} \succeq_{\mathcal{K}} H_2^{\texttt{happy}}$ holds. Let $\sigma = \{\texttt{X/a}\}$ a Skolem substitution for $H_2^{\texttt{happy}}$ with respect to $\mathcal{K} \cup H_1^{\texttt{happy}}$ and $\theta = \{\texttt{A/a}\}$ a ground substitution for $H_1^{\texttt{happy}}$. The condition (i) is immediately verified. The condition (ii) $\mathcal{K} \cup \{\texttt{famous(a)}\} \models \texttt{RICH(a)}$ is nothing else that a ground query answering problem in $\mathcal{SHIQ}$+log. It can be proved that the query `RICH(a)` can not be satisfied because the rule R1 is not applicable for `a`. Thus, $H_1^{\texttt{happy}} \not\succeq_{\mathcal{K}} H_2^{\texttt{happy}}$. Since $H_2^{\texttt{happy}} \not\succeq_{\mathcal{K}} H_1^{\texttt{happy}}$, the two hypotheses are incomparable under $\mathcal{K}$-subsumption. Conversely, it can be proved that $H_2^{\texttt{happy}} \succeq_{\mathcal{K}} H_3^{\texttt{happy}}$ but not viceversa.

Example 7. Let us consider the hypotheses

$H_1^{\texttt{LONER}}$ `LONER(A)` $\leftarrow$ `scientist(A)`
$H_2^{\texttt{LONER}}$ `LONER(X)` $\leftarrow$ `scientist(X),UNMARRIED(X)`

reported in Example 5 up to variable renaming. We want to check whether $H_1^{\texttt{LONER}} \succeq_{\mathcal{K}} H_2^{\texttt{LONER}}$ holds. Let $\sigma = \{\texttt{X/a}\}$ a Skolem substitution for $H_2^{\texttt{LONER}}$ with respect to $\mathcal{K} \cup H_1^{\texttt{LONER}}$ and $\theta = \{\texttt{A/a}\}$ a ground substitution for $H_1^{\texttt{LONER}}$. The condition (i) is immediately verified. The condition

$$\text{(ii) } \mathcal{K} \cup \{\texttt{scientist(a)}, \texttt{UNMARRIED(a)}\} \models \{\texttt{scientist(a)}\}$$

is a ground query answering problem in $\mathcal{SHIQ}$+log. It can be easily proved that all NM-models for $\mathcal{K} \cup \{\texttt{scientist(a)}, \texttt{UNMARRIED(a)}\}$ satisfy `scientist(a)`. Thus, $H_1^{\texttt{LONER}} \succeq_{\mathcal{K}} H_2^{\texttt{LONER}}$. The viceversa does not hold. Also it can be proved that $H_3^{\texttt{LONER}}$ is incomparable with both $H_1^{\texttt{LONER}}$ and $H_2^{\texttt{LONER}}$ under $\mathcal{K}$-subsumption.

It is straightforward to see that the decidability of $\mathcal{K}$-subsumption follows from the decidability of $\mathcal{SHIQ}$+log$^{\neg}$. It can be proved that $\succeq_{\mathcal{K}}$ is a quasi-order (i.e. it is a reflexive and transitive relation) for $\mathcal{SHIQ}$+log$^{\neg}$ rules, therefore the space of hypotheses can be searched by refinement operators.

[5] Since UNA does not necessarily hold in $\mathcal{SHIQ}$, the (Boolean) CQ/UCQ containment problem for $\mathcal{SHIQ}$ boils down to the (Boolean) CQ/UCQ answering problem.

4.2 The hypothesis coverage of observations

The definition of a **coverage relation** depends on the representation choice for observations. An observation $o_i \in O$ is represented as a couple $(p(\boldsymbol{a_i}), \mathcal{F}_i)$ where $\mathcal{F}_i$ is a set containing ground facts concerning the tuple of individuals $\boldsymbol{a_i}$. We assume $\mathcal{K} \cap O = \emptyset$.

Definition 2. *Let $H \in \mathcal{L}$ be a hypothesis, $\mathcal{K}$ a background knowledge and $o_i \in O$ an observation. We say that H* covers *o_i* under interpretations w.r.t. *$\mathcal{K}$ iff $\mathcal{K} \cup \mathcal{F}_i \cup H \models p(\boldsymbol{a_i})$.*

Note that the coverage test can be reduced to query answering in $\mathcal{SHIQ}$+log$^{\neg}$ KBs which in its turn can be reformulated as a satisfiability problem of the KB.

Example 8. The hypothesis $H_3^{\texttt{happy}}$ mentioned in Example 4 covers the observation $o_{\texttt{Mary}} = (\texttt{happy(Mary)}, \mathcal{F}_{\texttt{Mary}})$ because $\mathcal{K} \cup \mathcal{F}_{\texttt{Mary}} \cup H_3^{\texttt{happy}} \models \texttt{happy(Mary)}$. Indeed, all NM-models for $\mathcal{B} = \mathcal{K} \cup \mathcal{F}_{\texttt{Mary}} \cup H_3^{\texttt{happy}}$ satisfy:

- `famous(Mary)` (trivial!);
- $\exists$ `WANTS-TO-MARRY`$^-$.$\top$`(Mary)`, due to the axiom A1 and to the fact that both `RICH(Mary)` and `UNMARRIED(Mary)` hold in every model of $\mathcal{B}$;
- `happy(Mary)`, due to the above conclusions and to the rule R1. Indeed, since $\exists$`WANTS-TO-MARRY`$^-$.$\top$`(Mary)` holds in every model of $\mathcal{B}$, it follows that in every model there exists a constant `x` such that `WANTS-TO-MARRY(x,Mary)` holds in the model, consequently from rule R1 it follows that `happy(Mary)` also holds in the model.

Note that $H_3^{\texttt{happy}}$ does not cover the observations $o_{\texttt{Joe}} = (\texttt{happy(Joe)}, \mathcal{F}_{\texttt{Joe}})$ and $o_{\texttt{Paul}} = (\texttt{happy(Paul)}, \mathcal{F}_{\texttt{Paul}})$. More precisely, $\mathcal{K} \cup \mathcal{F}_{\texttt{Joe}} \cup H_3^{\texttt{happy}} \not\models \texttt{happy(Joe)}$ because `scientist(Joe)` holds in every model of $\mathcal{B} = \mathcal{K} \cup \mathcal{F}_{\texttt{Joe}} \cup H_3^{\texttt{happy}}$, thus making the rule R1 not applicable for {`X/Joe`}, therefore `RICH(Joe)` not derivable. Finally, $\mathcal{K} \cup \mathcal{F}_{\texttt{Paul}} \cup H_3^{\texttt{happy}} \not\models \texttt{happy(Paul)}$ because `UNMARRIED(Paul)` is not forced to hold in every model of $\mathcal{B} = \mathcal{K} \cup \mathcal{F}_{\texttt{Paul}} \cup H_3^{\texttt{happy}}$, therefore $\exists$`WANTS-TO-MARRY`$^-$.$\top$`(Paul)` is not forced by A1 to hold in every such model.

It can be proved that $H_1^{\texttt{happy}}$ covers $o_{\texttt{Mary}}$ and $o_{\texttt{Paul}}$, while $H_2^{\texttt{happy}}$ all the three observations.

Example 9. With reference to Example 5, the hypothesis $H_3^{\texttt{LONER}}$ does not cover the observation $o_{\texttt{Mary}} = (\texttt{LONER(Mary)}, \mathcal{F}_{\texttt{Mary}})$ because all NM-models for $\mathcal{B} = \mathcal{K} \cup \mathcal{F}_{\texttt{Mary}} \cup H_3^{\texttt{LONER}}$ do satisfy `famous(Mary)`. Note that it does not cover the observations $o_{\texttt{Paul}} = (\texttt{LONER(Paul)}, \mathcal{F}_{\texttt{Paul}})$ and $o_{\texttt{Joe}} = (\texttt{LONER(Joe)}, \mathcal{F}_{\texttt{Joe}})$ for analogous reasons. It can be proved that $H_2^{\texttt{LONER}}$ covers $o_{\texttt{Mary}}$ and $o_{\texttt{Joe}}$ while $H_1^{\texttt{LONER}}$ all three observations.

5 Related Work

Two ILP frameworks have been proposed so far that adopt a hybrid DL-CL representation for both hypotheses and background knowledge. The framework

proposed in [24] focuses on discriminant induction and adopts the ILP setting of learning from interpretations. Hypotheses are represented as CARIN-$\mathcal{ALN}$ non-recursive rules with a Horn literal in the head that plays the role of target concept. The coverage relation of hypotheses against examples adapts the usual one in learning from interpretations to the case of hybrid CARIN-$\mathcal{ALN}$ BK. The generality relation between two hypotheses is defined as an extension of generalized subsumption. Procedures for testing both the coverage relation and the generality relation are based on the existential entailment algorithm of CARIN. Following [24], Kietz studies the learnability of CARIN-$\mathcal{ALN}$, thus providing a pre-processing method which enables ILP systems to learn CARIN-$\mathcal{ALN}$ rules [13]. In [15], the representation and reasoning means come from $\mathcal{AL}$-log. Hypotheses are represented as constrained DATALOG clauses. Note that this framework is general, meaning that it is valid whatever the scope of induction is. The generality relation for one such hypothesis language is an adaptation of generalized subsumption to the $\mathcal{AL}$-log KR framework. It gives raise to a quasi-order and can be checked with a decidable procedure based on constrained SLD-resolution. Coverage relations for both ILP settings of learning from interpretations and learning from entailment have been defined on the basis of query answering in $\mathcal{AL}$-log. As opposite to [24], the framework has been partially implemented in an ILP system [16] that supports a variant of frequent pattern discovery where rich prior conceptual knowledge is taken into account in order to find patterns at multiple levels of description granularity.

Table 2. Comparison between ILP frameworks for DL-CL systems.

	Learning in Carin-$\mathcal{ALN}$ [24]	**Learning in $\mathcal{AL}$-log** [15]	**Learning in $\mathcal{SHIQ}$+log$^\neg$**
prior knowledge	CARIN-$\mathcal{ALN}$ KB	$\mathcal{AL}$-log KB	$\mathcal{SHIQ}$+log$^\neg$ KB
ontology lang.	$\mathcal{ALN}$	$\mathcal{ALC}$	$\mathcal{SHIQ}$
rule lang.	Horn clauses	DATALOG clauses	DATALOG$^\neg$ clauses
hypothesis lang.	CARIN-$\mathcal{ALN}$ non-recursive rules	constrained DATALOG clauses	$\mathcal{SHIQ}$+log$^\neg$ non-recursive rules
target predicate	Horn literal	DATALOG literal	$\mathcal{SHIQ}$/DATALOG literal
observations	interpretations	interpretations/implications	interpretations
induction	predictive	predictive/descriptive	predictive/descriptive
generality order	extension of [3] to CARIN-$\mathcal{ALN}$	extension of [3] to $\mathcal{AL}$-log	extension of [3] to $\mathcal{SHIQ}$+log$^\neg$
coverage test	CARIN-$\mathcal{ALN}$ query answering	$\mathcal{AL}$-log query answering	$\mathcal{SHIQ}$+log$^\neg$ query answering
ref. operators	no	downward	no
implementation	no	partially	no
application	no	yes	no

The ILP framework presented in this paper differs from [24] and [15] in several respects as summarized in Table 2, notably the following ones. First, it relies on a more expressive DL, i.e. $\mathcal{SHIQ}$. Second, it allows for inducing definitions for new DL concepts, i.e. rules with a $\mathcal{SHIQ}$ literal in the head. Third, it relies on a more expressive yet decidable CL, i.e. DATALOG$^\neg$. Forth, it adopts a tighter form of integration between the DL and the CL part, i.e. the weakly-safe one. Similarities also emerge from Table 2 such as the use of a *semantic* ordering for hypotheses in order to accommodate ontologies in ILP. Note that generalized subsumption

is chosen for adaptation in all three ILP frameworks because definite clauses, though enriched with DL and NAF literals, are still used.

6 Final Remarks

In this paper, we have proposed an ILP framework built upon $\mathcal{SHIQ}$+log$^{\neg}$. Indeed, well-known ILP techniques for induction have been reformulated in terms of the deductive reasoning mechanims of $\mathcal{DL}$+log. Notably, we have defined a decidable generality ordering, $\mathcal{K}$-subsumption, for $\mathcal{SHIQ}$+log$^{\neg}$ rules on the basis of the decidable algorithm NMSAT-$\mathcal{SHIQ}$+log. We would like to point out that the ILP framework proposed is suitable for inductive reasoning in the context of the Semantic Web for two main reasons. First, it adopts the DL which was the starting point for the design of the Web ontology language OWL. Second, it can deal with incomplete knowledge, thus coping with a more plausible scenario of the Web. Though the work presented in this paper can be considered as a feasibility study, it provides the principles for inductive reasoning in $\mathcal{SHIQ}$+log$^{\neg}$. We would like to emphasize that they will be still valid for any other upcoming decidable instantiation of $\mathcal{DL}$+log, provided that DATALOG$^{\neg}$ is still considered for the CL part.

The Semantic Web offers several use cases for rules among which we can choose in order to see our ILP framework at work. As next step towards any practice, we plan to define ILP algorithms starting from the ingredients identified in this paper. Tractable cases, e.g. the instantiation of $\mathcal{DL}$+log with DL-Lite (subset of $\mathcal{SHIQ}$), will be of major interest. Also we would like to investigate the impact of having DATALOG$^{\neg\vee}$ both in the language of hypotheses and in the language for the background theory. The inclusion of the nonmonotonic features of $\mathcal{SHIQ}$+log *full* will strengthen the ability of our ILP framework to deal with incomplete knowledge by performing an inductive form of commonsense reasoning. One such ability can turn out to be useful in the Semantic Web, and complementary to reasoning with uncertainty and under inconsistency. Finally, we would like to study the complexity of $\mathcal{K}$-subsumption.

References

1. F. Baader, D. Calvanese, D. McGuinness, D. Nardi, and P.F. Patel-Schneider, editors. *The Description Logic Handbook: Theory, Implementation and Applications*. Cambridge University Press, 2003.
2. A. Borgida. On the relative expressiveness of description logics and predicate logics. *Artificial Intelligence*, 82(1–2):353–367, 1996.
3. W. Buntine. Generalized subsumption and its application to induction and redundancy. *Artificial Intelligence*, 36(2):149–176, 1988.
4. S. Ceri, G. Gottlob, and L. Tanca. What you always wanted to know about datalog (and never dared to ask). *IEEE Transactions on Knowledge and Data Engineering*, 1(1):146–166, 1989.
5. L. De Raedt and L. Dehaspe. Clausal Discovery. *Machine Learning*, 26(2–3):99–146, 1997.

6. F.M. Donini, M. Lenzerini, D. Nardi, and A. Schaerf. $\mathcal{AL}$-log: Integrating Datalog and Description Logics. *Journal of Intelligent Information Systems*, 10(3):227–252, 1998.
7. T. Eiter, G. Gottlob, and H. Mannila. Disjunctive DATALOG. *ACM Transactions on Database Systems*, 22(3):364–418, 1997.
8. A.M. Frisch and A.G. Cohn. Thoughts and afterthoughts on the 1988 workshop on principles of hybrid reasoning. *AI Magazine*, 11(5):84–87, 1991.
9. M. Gelfond and V. Lifschitz. Classical negation in logic programs and disjunctive databases. *New Generation Computing*, 9(3/4):365–386, 1991.
10. B. Glimm, I. Horrocks, C. Lutz, and U. Sattler. Conjunctive query answering for the description logic $\mathcal{SHIQ}$. *Journal of Artificial Intelligence Research*, 31:151–198, 2008.
11. I. Horrocks, P.F. Patel-Schneider, and F. van Harmelen. From $\mathcal{SHIQ}$ and RDF to OWL: The making of a web ontology language. *Journal of Web Semantics*, 1(1):7–26, 2003.
12. I. Horrocks, U. Sattler, and S. Tobies. Practical reasoning for very expressive description logics. *Logic Journal of the IGPL*, 8(3):239–263, 2000.
13. J.-U. Kietz. Learnability of description logic programs. In S. Matwin and C. Sammut, editors, *Inductive Logic Programming*, volume 2583 of *Lecture Notes in Artificial Intelligence*, pages 117–132. Springer, 2003.
14. A.Y. Levy and M.-C. Rousset. Combining Horn rules and description logics in CARIN. *Artificial Intelligence*, 104:165–209, 1998.
15. F.A. Lisi. Building Rules on Top of Ontologies for the Semantic Web with Inductive Logic Programming. *Theory and Practice of Logic Programming*, 8(03):271–300, 2008.
16. F.A. Lisi and D. Malerba. Inducing Multi-Level Association Rules from Multiple Relations. *Machine Learning*, 55:175–210, 2004.
17. J.W. Lloyd. *Foundations of Logic Programming*. Springer, 2nd edition, 1987.
18. T.M. Mitchell. Generalization as search. *Artificial Intelligence*, 18:203–226, 1982.
19. S.-H. Nienhuys-Cheng and R. de Wolf. *Foundations of Inductive Logic Programming*, volume 1228 of *Lecture Notes in Artificial Intelligence*. Springer, 1997.
20. G.D. Plotkin. A note on inductive generalization. *Machine Intelligence*, 5:153–163, 1970.
21. R. Reiter. Equality and domain closure in first order databases. *Journal of ACM*, 27:235–249, 1980.
22. R. Rosati. Semantic and computational advantages of the safe integration of ontologies and rules. In F. Fages and S. Soliman, editors, *Principles and Practice of Semantic Web Reasoning*, volume 3703 of *Lecture Notes in Computer Science*, pages 50–64. Springer, 2005.
23. R. Rosati. $\mathcal{DL}$+log: Tight integration of description logics and disjunctive datalog. In P. Doherty, J. Mylopoulos, and C.A. Welty, editors, *Proc. of Tenth International Conference on Principles of Knowledge Representation and Reasoning*, pages 68–78. AAAI Press, 2006.
24. C. Rouveirol and V. Ventos. Towards Learning in CARIN-$\mathcal{ALN}$. In J. Cussens and A. Frisch, editors, *Inductive Logic Programming*, volume 1866 of *Lecture Notes in Artificial Intelligence*, pages 191–208. Springer, 2000.
25. C. Sakama. Nonmonotonic inductive logic programming. In T. Eiter, W. Faber, and M. Truszczynski, editors, *Logic Programming and Nonmonotonic Reasoning*, volume 2173 of *Lecture Notes in Computer Science*, pages 62–80. Springer, 2001.
26. M. Schmidt-Schauss and G. Smolka. Attributive concept descriptions with complements. *Artificial Intelligence*, 48(1):1–26, 1991.

A semantic stateless service description language

P. A. Bonatti and L. Sauro

Università di Napoli Federico II

Abstract. Complexity issues and the requirements on semantic web application in the Life Science domains recently motivated a few works on stateless service description languages [1, 5]. With stateless services, it is possible to reason about the semantic relationships between inputs and outputs, while keeping matchmaking and composition decidable. In this paper we extend the languages introduced in [1] and [5] with more general forms of composition and other constructs. We provide formal syntax and semantics and some preliminary results on the complexity of service comparison. These complexity results rely on hybrid formalisms involving both logic programming rules and description logics.

1 Introduction

The area of semantic web services is concerned with the declarative, knowledge based specification of web service semantics applied to service matchmaking (i.e., finding a service that matches a given specification), verification and automated composition. There is a conspicuous literature on the topic, enriched by several competing standards, such as OWL-S, WSMO, and WSDL-S.

When the semantic description involves dynamic behavioral aspects such as iterations, the tasks of matchmaking and composition easily become undecidable. This motivated a few works on stateless services [1, 5], that behave like functions or database queries. With stateless services, it is possible to move beyond a mere description of input and output types and capture the *relationships* between inputs and outputs, while keeping matchmaking and composition decidable. Stateless services are interesting because they are common in the domain of Life Sciences [5]. Moreover, they can be paired with a workflow language supporting procedural constructs like BPEL4WS with the purpose of supporting the dynamic binding of atomic activities.

In this paper we extend the languages introduced in [1] and [5] with more general forms of composition and other constructs. We provide formal syntax and semantics and some preliminary results on the complexity of service comparison, a basic reasoning task that underlies both matchmaking and composition (cf. [1]). These complexity results rely on hybrid formalisms involving both rules and description logics. The language we adopt admits a graphic presentation (that may be appreciated by users with limited programming skills) as well as textual representation that resembles relational query and programming languages enough to be familiar to programmers.

We start with some examples (Sec. 2) followed by a brief summary of description logic notions (Sec. 3). Then we formalize our service description logic language $\mathcal{SDL}_{full}$ (Sec. 4). Service comparison is reduced to an intermediate logic programming formulation and then to queries against description logic knowledge bases in Sec. 5, which allow to derive complexity results (Sec.6).

2 A running example

Services receive input messages and return output messages. Such messages are structured objects (as in WSDL), consisting of a set of attribute-value pairs, such as

$$\{\texttt{street="Via Toledo", numb=128}\}.$$

Following [1], we assume that services can be like queries, that is, a single input message may be mapped onto a *set* of homogeneously structured output messages. Formally this means that a service can be abstracted by any set of pairs $(m_{\mathsf{in}}, m_{\mathsf{out}})$, with multiple pairs sharing the same m_{in}.

Now assume an underlying ontology defines the concepts `Place`, `Map`, `Coord`, and `Address`, and that every `Place` has the attributes `hasAddr`, `hasMap`, `hasCoord`. In turn each address has the attributes `hasCity`, `hasStr` and `hasNum`. Consider a service *Mapservice* that takes input messages with attributes `city` and `street`, and returns the map of the surrounding area in a message with the single field `result`. *Mapservice* can be described in our language with the following expression:

select `result:=hasMap` **from all** `Place`
with `hasAddr.hasCity = city, hasAddr.hasStr = street .`

This description can be easily adapted to describe similar services. For example, a specialized map service that works only for southern cities can be described by defining a concept `SouthernCity` in the underlying ontology and restricting *Mapservice* with the expression:

Mapservice **restricted to** `SouthernCity(city) .`

Portals can be described with unions. Given two map services for Europe and China, called *Euromap* and *Chinamap*, a portal that covers both areas can be described by:

`union` (*Euromap, Chinamap*).

Intersections are supported, too. Now suppose that *Euromap* is more reliable than the generic *Mapservice*, then it may be preferable to use *Euromap* when possible. This can be done with conditionals (temporarily assume that *Euromap* and *Chinamap* have the same input message type as *Mapservice*, with the `city` field):

`if EuropeanCity(city)` **then** *Euromap* **else** *Mapservice* .

A relevant task is composition, our framework supports composition through *dataflow graphs* by which the output of some services can be fed as input to other services. For example, let *Addr2coor* be a service that takes `city` and `street` and returns the associated coordinates `lat` and `lon`; then let *Coor2map* be a service that returns the map associated to the given coordinates, called `latitude` and `latitude` by this service. The composition of these two services can be specified with the dataflow graph in Fig. 1. We support also a textual representation:

CompoundMap:
in `city, street`
out `result`
`C` := *Addr2coor*(in)
out := *Coor2map*(`latitude:=C.lat, longitude:=C.lon`).

In order to combine different services it may be necessary to adapt and restructure their inputs and outputs (e.g. consider the above example for conditionals when *Euromap* and *Chinamap* have different input message types). Here is an example of a variant of

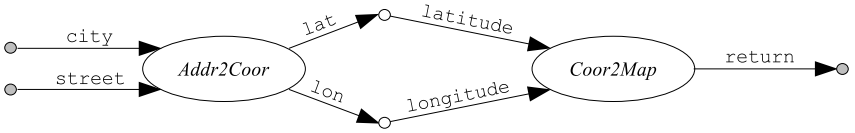


Fig. 1. A dataflow graph

Mapservice whose input city is forced to be `Naples` (a constant in the knowledge base), and whose output is renamed:

RestructuredMapServ:
in `street`
out `map`
`C` := *Mapservice* (`street:=`in`.street, city:=Naples`)
out `.map:=C.result.`

In general we allow a message element to be fed as an input to multiple other services, so dataflow graphs can be arbitrary DAGs. This was not allowed in [1]

Our framework allows to reason about different specifications. The basic reasoning task is *service comparison*, that given two service descriptions S_1 and S_2 checks whether all the input-output message pairs in the semantics of S_1 are also in the semantics of S_2; in that case we write $S_1 \sqsubseteq_{KB,\Sigma} S_2$, where KB is the underlying ontology and Σ contains the service definitions. By comparing services one may look for stronger or weaker services (cf. [1]). If *Addr2coor* and *Coor2map* are correctly specified (say, with `select` expressions), then our framework can verify that *CompoundMap* $\sqsubseteq_{KB,\Sigma}$ *Mapservice* and *Mapservice* $\sqsubseteq_{KB,\Sigma}$ *CompoundMap*, thereby concluding that in the absence of a direct implementation of *Mapservice*, an equivalent service can be obtained by composing the implementations of *Addr2coor* and *Coor2map* as specified by *CompoundMap* (dynamic service replacement). Service comparison can also be a basis for automated composition that, however, lies beyond the scope of this paper.

Syntactically speaking, the service description language illustrated above lies somewhere in between relational algebra and a programming language. A major difference with respect to both is that descriptions are linked to an ontology, so it is possible to distinguish—say—a hash table that associates people with their age from another hash table (with the same implementation) that associates people with their credit card number. Clearly, such differences are crucial for tasks such as service discovery and dynamic binding of workflow activities to services. Procedural constructs cover assignments and conditionals; only iterations are not supported, and this has a few advantages: (i) the main reasoning tasks are decidable, (ii) the language is easier to use for people with no programming background.

3 Preliminaries

The vocabulary of the description logics we deal with in this paper consists of the following pairwise disjoint countable sets of symbols: a set of *atomic concepts* At, a set of individual names In, and a set of *atomic roles* $\mathsf{A_R}$, with a distinguished subset of names $\mathsf{A_{tR}} \subseteq \mathsf{A_R}$ denoting *transitive roles*.

A *role* is either an expression P or P^-, where $P \in \mathsf{A_R}$. Let R range over roles. The set of *concepts* is the smallest superset of At such that if C, D are concepts, then $\top$, $\neg C$, $C \sqcap D$, $\exists.C$, and $\exists^{\leq n} R.C$ are concepts.

Semantics is based on interpretations of the form $\mathcal{I} = \langle \Delta^{\mathcal{I}}, \cdot^{\mathcal{I}} \rangle$ where $\Delta^{\mathcal{I}}$ is a set of *individuals* and $\cdot^{\mathcal{I}}$ is an interpretation function mapping each $A \in \mathsf{At}$ on some $A^{\mathcal{I}} \subseteq \Delta^{\mathcal{I}}$, each $a \in \mathsf{In}$ on some $a^{\mathcal{I}} \in \Delta^{\mathcal{I}}$, and each $R \in \mathsf{A_R}$ on some $R^{\mathcal{I}} \subseteq \Delta^{\mathcal{I}} \times \Delta^{\mathcal{I}}$. Moreover, if $R \in \mathsf{A_{tR}}$, then $R^{\mathcal{I}}$ is transitive. The meaning of inverse roles is $(R^-)^{\mathcal{I}} = \{\langle y, x\rangle \mid \langle x, y\rangle \in R^{\mathcal{I}}\}$. Next we define the meaning of compound concepts. By $\sharp S$ we denote the cardinality of S.

$$
\begin{aligned}
A_\rho^{\mathcal{I}} &= A^{\mathcal{I}} \qquad (A \in \mathsf{At}) \qquad & \top^{\mathcal{I}} &= \Delta^{\mathcal{I}} \\
(\neg C)_\rho^{\mathcal{I}} &= \Delta^{\mathcal{I}} \setminus C_\rho^{\mathcal{I}} & (C \sqcap D)_\rho^{\mathcal{I}} &= C_\rho^{\mathcal{I}} \cap D_\rho^{\mathcal{I}} \\
(\exists R.C)_\rho^{\mathcal{I}} &= \left\{x \mid \exists y.\langle x, y\rangle \in R^{\mathcal{I}} \wedge y \in C_\rho^{\mathcal{I}}\right\} & & \\
(\exists^{\leq n} R.C)_\rho^{\mathcal{I}} &= \left\{x \mid \sharp\{y \mid \langle x, y\rangle \in R^{\mathcal{I}} \wedge y \in C_\rho^{\mathcal{I}}\} \leq n\right\} . & &
\end{aligned}
$$

Other standard constructs ($\forall R.C,\ \bot, \sqcup$) can be derived from the above concepts.

A *general concept inclusion* (GCI) is an expression $C \sqsubseteq D$ where C and D are concepts. A *role inclusion* is an expression $R_1 \sqsubseteq R_2$ where R_1 and R_2 are roles. An *assertion* is an atom like $A(a)$ or $P(a, b)$ where $A \in \mathsf{At}$, $R \in \mathsf{A_R}$, and $\{a, b\} \in \mathsf{In}$. A *TBox* is a set of GCIs; a *role hierarchy* is a set of role inclusions; an *ABox* is a set of assertions. Finally, a DL knowledge base (DL KB) is a triple $\langle \mathcal{T}, \mathcal{H}, \mathcal{A}\rangle$ consisting of a TBox, a role hierarchy and an ABox.

An interpretation $\mathcal{I}$ satisfies a (concept or role) inclusion $E_1 \sqsubseteq E_2$ iff $E_1^{\mathcal{I}} \subseteq E_2^{\mathcal{I}}$. Moreover $\mathcal{I}$ satisfies an assertion $A(a)$ (resp. $P(a, b)$) iff $a^{\mathcal{I}} \in A^{\mathcal{I}}$ (resp. $(a^{\mathcal{I}}, b^{\mathcal{I}}) \in P^{\mathcal{I}}$). A *model* of a DL KB is any $\mathcal{I}$ that satisfies all the inclusions and the assertions of the KB.

The above description logic is known as $\mathcal{SHIQ}_{R+}$. By disallowing transitive roles we get $\mathcal{SHIQ}$. By disallowing $\exists^{\leq n} R.C$, transitive roles and role hierarchies one gets the logic $\mathcal{ALCI}$. $\mathcal{ALC}$ is obtained by further dropping inverse roles. The logic $\mathcal{EL}$ supports only $\sqcap$, $\exists R.\top$, and GCIs built from these constructs.

Moreover, there exists a rather different extension of $\mathcal{ALC}$ called $\mathcal{DLR}$, supporting n-ary relations ($n > 2$) that we will mention in the following but we do not report here due to space limitations. Its definition and relevant results can be found in [3].

4 Syntax and semantics of $\mathcal{SDL}_{full}$

Our service description language, called $\mathcal{SDL}_{full}$, extends the DL vocabulary with an infinite supply of constants $\mathsf{N_c}$, service names $\mathsf{N_s}$, and message attribute names $\mathsf{N_a}$. $\mathcal{SDL}_{full}$ describes functional and knowledge-based aspects of web-services. Therefore, as usual functional programming languages, it does not define a service as a set of state variables and a sequence of statements which update them, but as the functional composition of stateless expressions that have to be evaluated.

Definition 1. *The language of* service expressions *is the least set* Expr *containing:*

- *(service calls) all* $S \in \mathsf{N_s}$*;*
- *(set operators) all expressions* $op(E_1, \ldots, E_n)$ *such that* $\{E_1, \ldots, E_n\} \subseteq$ Expr *and* $op \in \{\texttt{union}, \texttt{intersection}\}$*;*

– *(conditionals) all expressions* `if` L `then` E_1[`else` E_2] *(the else clause is optional) such that*
  - $\{E_1, E_2\} \subseteq$ Expr*, and*
  - L *is a list of* conditions *of the form* $t = u$, $t \neq u$, $A(t)$, *or* $\neg A(t)$, *where* $\{t, u\} \subseteq \mathsf{N_c} \cup \mathsf{N_a}$
– *(selections) all expressions* `select` $a_1 := r_1, \ldots a_n := r_n$ `from all` D `with` L*, such that*
  - $a_i \in \mathsf{N_a}$ $(1 \leq i \leq n)$*;*
  - r_i *is a role path (in the language of the underlying ontology)* $(1 \leq i \leq n)$*;*
  - D *is a concept (in the language of the underlying ontology);*
  - L *is a list of* bindings $p_i = t_i$ $(1 \leq i \leq m)$ *where each* p_i *is a role path (in the language of the underlying ontology), and* $t_i \in \mathsf{N_c} \cup \mathsf{N_a}$*;*
– *(message restructuring) all expressions* $a_1 := t_1, \ldots a_n := t_n$ *such that* $a_i \in \mathsf{N_a}$ *and* $t_i \in \mathsf{N_a} \cup \mathsf{N_c}$ $(1 \leq i \leq n)$*;*
– *(restrictions) all expressions* E `restricted to` L *such that* L *is a list of conditions (see conditionals above).*

A service consists of a dataflow graph which *evaluates* data by means of *functional* nodes. Each functional node represents a stateless expression which may have multiple inputs and outputs denoted by parameter names. Edges in a dataflow graph are used to connect the output of a functional node with the inputs of (possibly many) other functional nodes. In order to specify which output is connected to which input, edges are also labeled with attribute names and, as the inputs and the outputs of different expressions may be labeled with different parameter names, edges do not connect directly two functional nodes, but connect functional nodes with *parameter* nodes that are intended to fix name mismatches.

Dataflow graphs are defined as follows:

Definition 2. *A* dataflow graph *with name* S *is a tuple* $\langle S, N_S, E_S, name_S, expr_S \rangle$ *where*

– $S \in \mathsf{N_s}$*;*
– N_S *is a finite set of nodes, partitioned into* functional *and* parameter *nodes, denoted by* $\mathsf{fun}(N_S)$ *and* $\mathsf{par}(N_S)$*, respectively;*
– E_S *is a finite set of edges;* $E_S \subseteq (\mathsf{fun}(N_S) \times \mathsf{par}(N_S)) \cup (\mathsf{par}(N_S) \times \mathsf{fun}(N_S))$*;*
– $name_S : \mathsf{par}(N_S) \cup E_S \to \mathsf{N_a}$ *is a labelling function;*
– $expr_S : \mathsf{fun}(N_S) \to$ Expr *is a labelling function.*

Moreover, dataflow graphs are required to be directed acyclic graphs (DAGs).

The parameter nodes with no incoming edges (resp. no outgoing edges) will be called the *input nodes* (resp. *output nodes*) of the graph. In Fig. 1, ovals and small circles represent functional and parameter nodes, respectively; input and output nodes are colored in gray.

The *dependency graph* of a set of dataflow graphs Σ is $\langle \Sigma, E \rangle$, where E is the set of all pairs (G_1, G_2) such that the name of G_2 occurs in the label of some functional node in G_1. We say that Σ is *acyclic* if its dependency graph is.

Definition 3. *A* service specification Σ *is a finite, acyclic set of dataflow graphs with mutually different names.*

Edge labels should match the input/output message attributes of the service expressions labeling functional nodes. This requirement is formalized in terms of typing. In this paper we only deal with a structural form of typing (centred around message attribute names); the problem of ensuring–say—that the connected input/output attributes `lat` and `latitude` in Fig. 1 belong respectively to two "compatible" concepts C_1 and C_2 such that $C_1 \sqsubseteq C_2$ has already been tackled in the literature (including [5]). We will deal with it in the full paper.

Definition 4. *A (message)* type *is a finite set* $T \subseteq \mathsf{N_a}$

Definition 5. *The* input type of a dataflow graph $G = \langle S, N_S, E_S, name_S, expr_S \rangle$ *with respect to a specification* Σ *is the set*

$$\mathsf{in}_\Sigma(G) = \{name_S(n) \mid n \textit{ is an input node of } G\}\,.$$

The output type of a dataflow graph $G = \langle S, N_S, E_S, name_S, expr_S \rangle$ *with respect to a specification* Σ *is the set*

$$\mathsf{out}_\Sigma(G) = \{name_S(n) \mid n \textit{ is an output node of } G\}\,.$$

Definition 6. *The* input type of a service expression E *with respect to a specification* Σ*, denoted by* $\mathsf{in}_\Sigma(E)$*, is recursively specified as follows:*

- *if* $E = S \in \mathsf{N_s}$*, then* $\mathsf{in}_\Sigma(E)$ *equals* $\mathsf{in}_\Sigma(G)$ *where* G *has name* S*;*
- $\mathsf{in}_\Sigma(op(E_1, E_2)) = \mathsf{in}_\Sigma(E_1) \cup \mathsf{in}_\Sigma(E_2)$*;*
- $\mathsf{in}_\Sigma(\texttt{if}\, C\, \texttt{then}\, E_1\, \texttt{else}\, E_2) = \mathsf{in}_\Sigma(E_1) \cup \mathsf{in}_\Sigma(E_2) \cup \{a \in \mathsf{N_a} \mid a \textit{ occurs in } C\}$*;*
- $\mathsf{in}_\Sigma(\texttt{select}\, A\, \texttt{from all}\, D\, \texttt{with}\, R) = \{a \in \mathsf{N_a} \mid a \textit{ occurs in } R\}$*;*
- $\mathsf{in}_\Sigma(a_1 := t_1, \ldots a_n := t_n) = \mathsf{N_a} \cap \{t_1, \ldots t_n\}$.

The output type of a service expression E *with respect to a specification* Σ*, denoted by* $\mathsf{out}_\Sigma(E)$*, is recursively specified as follows:*

- *if* $E = S \in \mathsf{N_s}$*, then* $\mathsf{out}_\Sigma(E)$ *equals* $\mathsf{out}_\Sigma(G)$ *where* G *has name* S*;*
- $\mathsf{out}_\Sigma(op(E_1, E_2)) = \mathsf{out}_\Sigma(E_1) \cap \mathsf{out}_\Sigma(E_2)$*;*
- $\mathsf{out}_\Sigma(\texttt{if}\, C\, \texttt{then}\, E_1\, \texttt{else}\, E_2) = \mathsf{out}_\Sigma(E_1) \cap \mathsf{out}_\Sigma(E_2)$*;*
- $\mathsf{out}_\Sigma(\texttt{select}\, a_1 := r_1, \ldots a_n := r_n\, \texttt{from all}\, D\, \texttt{with}\, R) = \{a_1, \ldots a_n\}$*;*
- $\mathsf{out}_\Sigma(a_1 := t_1, \ldots a_n := t_n) = \{a_1, \ldots a_n\}$.

About the above definition: Intuitively, *all* input parameters have to be supplied in order to call a service; therefore if the components of a compound service have different input types, then the compound service must take their *union* to be sure that all component services can be invoked. Symmetrically, the only outputs one can count on are those returned by all the component services; this is why intersection is used here.

Definition 7. *A specification* Σ *is* well-typed *iff for all dataflow graphs* $\langle S, N_S, E_S, name_S, expr_S \rangle \in \Sigma$*, and for all functional nodes* $k \in \mathsf{fun}(N_S)$*,*

- $\mathsf{in}(k)$ *equals the set of labels of the incoming edges of* k*;*
- $\mathsf{out}(k)$ *contains the set of labels of the outgoing edges of* k.

From now on we assume that all service specifications are well-typed unless stated otherwise.

The semantics of service expressions and dataflow graphs is defined in terms of *worlds* that specify the extension of concepts and roles, as well as the behavior of each service. From a semantic perspective, a message is a partial function defined over the message's attributes, that returns for each attribute its value.

Definition 8. *A* Δ-message *is a partial function* $m : \mathsf{N_a} \to \Delta$.

The message's range Δ will sometimes be omitted when irrelevant or obvious.

Now a world is simply a combination of a DL interpretation (that interprets the terms defined in the underlying ontology) plus an interpretation of service names (i.e. atomic services).

Definition 9. *A* world *is a tuple* $\mathcal{W} = \langle \Delta^{\mathcal{W}}, \cdot^{\mathcal{W}}, [\![\cdot]\!]^{\mathcal{W}} \rangle$ *such that*

- $\langle \Delta^{\mathcal{W}}, \cdot^{\mathcal{W}} \rangle$ *is an interpretation of the knowledge base;*
- $[\![\cdot]\!]^{\mathcal{W}}$ *maps every service name* $S \in \mathsf{N_s}$ *on a set* $[\![S]\!]^{\mathcal{W}}$ *of* $\Delta^{\mathcal{W}}$*-message pairs.*

To ensure that service name evaluation reflects the given service specification, we have to specify the semantics of the terms and expressions used in dataflow graph labels.

Definition 10. *The* evaluation $t^{\mathcal{W}}(m)$ *of a* term $t \in \mathsf{N_c} \cup \mathsf{N_a}$ *with respect to a world* $\mathcal{W}$ *and a message* m*, is* $m(t)$ *if* $t \in \mathsf{N_a}$*, and* $t^{\mathcal{W}}$ *otherwise.*

Definition 11. *The* evaluation $E^{\mathcal{W}}(m)$ *of a* service expression E *with respect to a world* $\mathcal{W}$ *and a message* m *is recursively defined as follows:*

- *if* $E = S \in \mathsf{N_s}$*, then* $E^{\mathcal{W}}(m) = \{m' \mid (m, m') \in [\![S]\!]^{\mathcal{W}}\}$*;*
- $\texttt{union}(E_1, E_2)^{\mathcal{W}}(m) = E_1^{\mathcal{W}}(m) \cup E_2^{\mathcal{W}}(m)$*;*
- $\texttt{intersection}(E_1, E_2)^{\mathcal{W}}(m) = E_1^{\mathcal{W}}(m) \cap E_2^{\mathcal{W}}(m)$*;*
- $(\texttt{if}\, C\, \texttt{then}\, E_1\, \texttt{else}\, E_2)^{\mathcal{W}}(m) = E_1^{\mathcal{W}}(m)$ *if* $C^{\mathcal{W}}(m)$ *is true,* $E_2^{\mathcal{W}}(m)$ *otherwise; moreover,* $C^{\mathcal{W}}(m)$ *is true iff*
  - *for all* $t \odot u$ *in* C*,* $t^{\mathcal{W}}(m) \odot u^{\mathcal{W}}(m)$ *holds* $(\odot \in \{=, \neq\})$*,*
  - *and for all literals* $A(t)$ *and* $\neg B(u)$ *in* C*,* $t^{\mathcal{W}}(m) \in A^{\mathcal{W}}$ *and* $u^{\mathcal{W}}(m) \notin B^{\mathcal{W}}$*;*
- $(\texttt{select}\, a_1 := r_1, \ldots a_n := r_n\, \texttt{from all}\, D\, \texttt{with}\, R)^{\mathcal{W}}(m)$ *is the set of all* m' *such that, for some* $x \in D^{\mathcal{W}}$*,*
  - *for all* $r \odot t$ *in* R *there exists* $y \in r^{\mathcal{W}}(x)$ *such that* $y \odot t^{\mathcal{W}}(m)$ *holds* $(\odot \in \{=, \neq\})$*;*
  - $m'(a_i) \in r_i^{\mathcal{W}}(x)$ $(1 \leq i \leq n)$*;* m' *is undefined in every other case;*
- $(a_1 := t_1, \ldots a_n := t_n)^{\mathcal{W}}(m) = \{m'\}$ *where the domain of* m' *is* $a_1, \ldots a_n$ *and* $m'(a_i) = m(t_i)$ $(1 \leq i \leq n)$*.*

The evaluation of service compositions (i.e. dataflow graphs) is defined in a declarative way: each parameter node must be assigned a value (an element of $\Delta^{\mathcal{W}}$) in a way that is compatible with the input-output behavior of each functional node:

Definition 12. *The evaluation* $[\![G]\!]^{\mathcal{W}}$ *of a graph* $G = \langle S, N_S, E_S, name_S, expr_S \rangle$ *w.r.t.* $\mathcal{W}$ *is the set of all* $\Delta^{\mathcal{W}}$*-message pairs* (m_{in}, m_{out}) *such that for some function* $\sigma : \mathsf{par}(N_S) \to \Delta^{\mathcal{W}}$*, the following conditions hold:*

– *for all input nodes* $n \in N_S$, $m_{in}(name_S(n)) = \sigma(n)$;
– *for all output nodes* $n \in N_S$, $m_{out}(name_S(n)) = \sigma(n)$;
– m_{in} *and* m_{out} *are undefined for every other attribute name;*
– *for all* $n \in \mathsf{fun}(N_S)$, *it must hold that* $m^n_{out} \in expr_S(n)^{\mathcal{W}}(m^n_{in})$, *where* m^n_{in} *and* m^n_{out} *are defined as follows: for all* $a \in \mathsf{N_a}$,
  - *if there exists an edge* (n', n) *with* $name_S(n', n) = a$, *let* $m^n_{in}(a) = \sigma(n')$,
  - *if there exists an edge* (n, n'') *with* $name_S(n, n'') = a$, *let* $m^n_{out}(a) = \sigma(n'')$,
  - m^n_{in} *and* m^n_{out} *are undefined for all other inputs.*

Definition 13. *A world* $\mathcal{W}$ *is a* model *of a specification* Σ *with respect to a knowledge base* KB *iff*

1. $\langle \Delta^{\mathcal{W}}, \cdot^{\mathcal{W}} \rangle$ *is a model of* KB;
2. *for all names* S *of a dataflow graphs* $G \in \Sigma$, $[\![S]\!]^{\mathcal{W}} = [\![G]\!]^{\mathcal{W}}$.

If $\mathcal{W}$ is a model of Σ, then it is not hard to see that since Σ is acyclic (by definition), $[\![\cdot]\!]^{\mathcal{W}}$ is uniquely determined by $\langle \Delta^{\mathcal{W}}, \cdot^{\mathcal{W}} \rangle$ (i.e. service specifications are deterministic).

The next definition specifies when a service S_1 is a *weakening* of S_2 (equivalently, S_2 is a *strengthening* of S_1) [1]. These relations are the basis for service comparison.

Definition 14. $S_1 \sqsubseteq_{KB,\Sigma} S_2$ *iff for all models* $\mathcal{W}$ *of* Σ *w.r.t.* KB, $[\![S_1]\!]^{\mathcal{W}} \subseteq [\![S_2]\!]^{\mathcal{W}}$.

Roughly speaking, if S_2 is a strengthening of S_1, then for any given input, S_2 returns more answers than S_1. See [1] for a discussion of the different applications of strengthening and weakening in our reference scenarios.

5 Service comparison

Definition 15. *The* service comparison problem *is defined as follows: given* KB, Σ, *and two service names* S_1 *and* S_2, *decide whether* $S_1 \sqsubseteq_{KB,\Sigma} S_2$.

By translating service specifications into logic programming rules, service subsumption checking can be reduced to containment of unions of conjunctive queries (UCQ) against DL knowledge bases. In turn, this problem can be reduced to the evaluation of UCQs against DL knowledge bases.

5.1 Rules and queries

Consider rules like $A \leftarrow L_1, \ldots, L_n$ where A is a logical atom, each L_i is a literal (i.e. either an atom or a negated atom), possibly of the form $t = u$ or $t \neq u$. As usual, let $\mathsf{head}(r) = A$ and $\mathsf{body}(r) = \{L_1, \ldots, L_n\}$. *We restrict our attention to function-free rules only*: terms will be restricted to constants in In and variables.

The predicates in $\mathsf{body}(r)$ may be defined in a DL knowledge base, i.e. unary and binary predicates may belong to At and $\mathsf{A_R}$, respectively. If *all* the predicates occurring in $\mathsf{body}(r)$ belong to At and $\mathsf{A_R}$ and $\mathsf{body}(r)$ contains no occurrences of $\neg$, then we call r a *conjunctive query* (CQ). A *union of conjunctive queries* (UCQ) is a set of CQs having the same predicate name in the head. We add superscripts $\neq$, $\neg$ if the corresponding symbol

may occur in $\mathsf{body}(r)$; for example $\text{UCQ}^\neg$ denotes the unions of conjunctive queries that may contain negative literals in the body.

Let $\mathcal{P}$ be a set of rules and $\mathcal{I}$ be an interpretation. Let an *$\mathcal{I}$-substitution* be a substitution that replaces each constant a by $a^\mathcal{I}$, and each variable with an element of $\Delta^\mathcal{I}$. $\mathcal{I}$-substitutions are a useful tool for defining the semantics of rules and queries.

Usually queries are evaluated against a knowledge base, and the answer is restricted to the individual constants that explicitly occur in the ABox (e.g. see [8]). In particular, a tuple $\boldsymbol{c}$ of constants is a *certain answer* of a CQ r against a DL KB $\mathcal{K}$ iff

- the constants in $\boldsymbol{c}$ occur in $\mathcal{K}$; moreover, for some substitution σ defined on the variables of $\mathsf{head}(r)$,
- $\mathsf{head}(r\sigma)$ has the form $p(\boldsymbol{c})$;
- for all models $\mathcal{I}$ of $\mathcal{K}$, there exists an $\mathcal{I}$-substitution θ such that every literal in $\mathsf{body}(r\sigma\theta)$ is *satisfied* by $\mathcal{I}$, that is,
  - for all $A(d)$ (resp. $\neg A(d)$) in $\mathsf{body}(r'\sigma\theta)$, $d \in A^\mathcal{I}$ (resp. $d \notin A^\mathcal{I}$);
  - for all $P(d,e)$ (resp. $\neg P(d,e)$) in $\mathsf{body}(r'\sigma\theta)$, $(d,e) \in P^\mathcal{I}$ (resp. $(d,e) \notin P^\mathcal{I}$);
  - all literals $d = e$ and $d \neq e$ in $\mathsf{body}(r'\sigma\theta)$ are true.

The set of all certain answers of a CQ r against $\mathcal{K}$ will be denoted by $\mathsf{c\_ans}(r,\mathcal{K})$. For a UCQ Q, let $\mathsf{c\_ans}(r,\mathcal{K}) = \bigcup_{r\in Q} \mathsf{c\_ans}(r,\mathcal{K})$.

In this paper, we will also query the *models* of a knowledge base and introduce what we call *unrestricted answers*, that are built from the domain elements of the models.[1] This definition applies to all sets of rules (not only CQs and UCQs).

The *$\mathcal{I}$-reduct of $\mathcal{P}$* , $\mathcal{P}^\mathcal{I}$, is the set of all rules r such that for some $r' \in \mathcal{P}$ and some $\mathcal{I}$-substitution σ,

- all literals belonging to $\mathsf{body}(r'\sigma)$ whose predicate is in $\mathsf{At} \cup \mathsf{A_R} \cup \{=, \neq\}$ are satisfied by $\mathcal{I}$;
- r is obtained from $r'\sigma$ by removing from $\mathsf{body}(r'\sigma)$ all the literals whose predicate is in $\mathsf{At} \cup \mathsf{A_R} \cup \{=, \neq\}$.

Note that the $\mathcal{I}$-reduct of a UCQ is always a set of facts.

We will denote by $\mathsf{lm}(\mathcal{P}^\mathcal{I})$ the least Herbrand model of $\mathcal{P}^\mathcal{I}$. The *unrestricted answer to a predicate p in $\mathcal{P}$* against $\mathcal{I}$ is $\mathsf{u\_ans}(p,\mathcal{P},\mathcal{I}) = \{\boldsymbol{c} \mid p(\boldsymbol{c}) \in \mathsf{lm}(\mathcal{P}^\mathcal{I})\}$.

5.2 The reduction

We proceed by illustrating the tranlation of service specifications into logic programs. Syntactically, such programs are like queries, but have the unrestricted semantics, like our service descriptions; so they provide a nice intermediate step for the complete reduction of service comparison to certain answers. In order to simplify the presentation, we assume that service specifications are normalized by replacing subexpressions with new services,

[1] This notion differs from the many hybrid combinations of rules and DLs (see [7] for a survey). The latter are still rather close to querying DL KBs and their answers are restricted to the constants occurring in the rules or in the KB. Moreover, the purpose is different: those combination are supposed to be knowledge representation formalisms, while our semantics is merely a technical device to link service comparison to query answering against DL KBs.

so that *no constructs are nested* (all subexpressions are service names). We use further service names to guarantee that *if a dataflow graph has more than one functional node, then all nodes are labelled with service names only*. Finally, we assume that message attributes are renamed so that *different functional nodes never share any message attribute name*. Clearly, the above normalizations take polynomial time.

Then for each service name S defined in the specification, we define an atom $p_S(X_{f_1}, \ldots, X_{f_m}, Y_{g_1}, \ldots, Y_{g_n})$, where p_S is a fresh predicate symbol, and $f_1, \ldots, f_m$ (resp. $g_1, \ldots, g_n$) is the lexicografic ordering of $\mathsf{in}(S)$ (resp. $\mathsf{out}(S)$). We denote the above atom by H_S.

Now each service S whose dataflow graph has multiple functional nodes with labels $S_1, \ldots, S_n$, can be translated into one rule $(H_S \leftarrow H_{S_1}, \ldots, H_{S_n})\sigma$, where the substitution σ unifies all variables Y_{g_i} and X_{f_j} such that some parameter node has an incoming edge labelled with g_i and an outgoing edge labelled f_j.

Next consider an S whose dataflow has a single functional node labelled E. If E is $\texttt{union}(S_1, \ldots, S_n)$ then S can be translated into n rules $H_S \leftarrow H_{S_i}$ $(1 \leq i \leq n)$. Symmetrically, if $E = \texttt{intersection}(S_1, \ldots, S_n)$ then S can be translated into one rule $H_S \leftarrow H_{S_1}, \ldots, H_{S_n}$.

When $E = \texttt{if}\, c_1, \ldots, c_n\, \texttt{then}\, S_1\, \texttt{else}\, S_2$, S is translated into the rules $H_S \leftarrow [c_1], \ldots, [c_n], S_1$ and $H_S \leftarrow [\bar{c}_i], S_2$ $(1 \leq i \leq n)$. Here each c_i is a condition and $[c_i]$ denotes its tranlation; $\bar{c}_i$ denotes the complement of c_i, e.g. if c_i is $x = y$ then $\bar{c}_i$ is $x \neq y$; if $c_i = A(x)$ then $\bar{c}_i = \neg A(x)$. The translation $[c_i]$ consists in turning each message attribute f into the corresponding variable X_f.

The translation of $\texttt{select}\, a_1 := r_1, \ldots, a_n := r_n\, \texttt{from all}\, A\, \texttt{with}\, p_1 = t_1, \ldots$ $p_m = t_m$ is $H_S \leftarrow A(Z), [a_1 := r_1], \ldots, [a_n := r_n], [p_1 = t_1], \ldots, [p_m = t_m]$, where Z is a fresh variable. Each $[a_i := r_i]$ consists of the translation of the role path r_i into a conjunction of binary atoms (using fresh variables at the intermediate steps), plus the atom $Y_{a_i} = V$, where V is the last variable introduced in the translation of r_i. Similarly, each $[p_i = t_i]$ consists of the translation of the role path p_i plus $u = V$, where V is the last variable introduced in the translation of p_i, and $u = t_i$ if $t_i \in \mathsf{N_c}$, otherwise (i.e. if $t_i \in \mathsf{N_a}$) $u = X_{t_i}$.

Example 1. In our running example, a condition like `hasAddr.hasStr=street` is translated into $\texttt{hasAddr}(Z, V_1),\ \texttt{hasStr}(V_1, V_2),\ X_{\texttt{street}} = V_2$, where V_1, V_2 are new variables.

Due to space limitations we omit the (straightforward) translation of message restructuring and restrictions.

Let us denote the translation of a specification Σ with P_Σ. The above translation is pretty natural and it is not hard to see that it preserves the meaning of the given normal specification under unrestricted query evaluation, as stated by the following theorem.

Theorem 1. *Let Σ be a normalized service specification and let P_Σ be its translation. Let S be the name of a graph $G \in \Sigma$ and $f_1, \ldots, f_m$ (resp. $g_1, \ldots, g_n$) be the lexicographic ordering of $\mathsf{in}(S)$ (resp. $\mathsf{out}(S)$).*

Then for all models $\mathcal{W}$ of Σ w.r.t. KB, $(t_1, \ldots, t_m, u_1, \ldots, u_n) \in \mathsf{u\_ans}(p_S, P_\Sigma, \mathcal{W})$ iff for some message pair $(m, m') \in [\![S]\!]^{\mathcal{W}}$, $m(f_i) = t_i$ and $m'(g_j) = u_j$ $(1 \leq i \leq m,\ 1 \leq j \leq n)$.

The above result can be reformulated in terms similar to query containment. For all predicates p_{S_1} and p_{S_2}, let $p_{S_1} \subseteq_{KB,\Sigma} p_{S_2}$ iff for all models $\mathcal{W}$ of Σ w.r.t. KB, $\mathsf{u\_ans}(p_{S_1}, P_\Sigma, \mathcal{W}) \subseteq \mathsf{u\_ans}(p_{S_2}, P_\Sigma, \mathcal{W})$.

Corollary 1. *For all normalized specifications Σ, $S_1 \sqsubseteq_{KB,\Sigma} S_2$ iff $p_{S_1} \subseteq_{KB,\Sigma} p_{S_2}$.*

6 Complexity results

In this section we exploit Theorem 1 and the many recent complexity results on certain query answers against DL knowledge bases to derive a preliminary set of complexity bounds for our service description language.

In order to illustrate decidable cases and complexity sources, we introduce a uniform notation for the fragments of our service description language $\mathcal{SDL}_{full}$:

- $\mathcal{SD}$ restricts the language by forbidding `union`, `else`, negative conditions (such as $r \neq t$ and $\neg A(t)$), and equality within conditions (equality is allowed in the `with` clause of selections);
- superscripts u and e stand for `union` and `else`, respectively; when they are present, the language supports the corresponding constructs;
- similarly, superscripts $=$, $\neq$ and $\neg$ stand for conditions with equalities, disequalities and concept complements, respectively;
- the superscript k imposes that the maximum nesting level of `union` and `else` is bounded by a constant k.

For example, $\mathcal{SD}^{u,\neq}$ stands for the sublanguage of $\mathcal{SDL}_{full}$ supporting `union` and conditions with disequalities, but neither `else` nor negative conditions like $\neg A(t)$. By $\mathcal{SD}^{k,u,\neq}$ we denote a similar language, where the nesting level of `union` is bounded by a constant k.

In this preliminary paper, we adopt the following reduction to obtain a first set of decidability results and complexity upper bounds:

1. Service comparison in Σ is reduced to unrestricted answer containment in P_Σ by Theorem 1; note that P_Σ can be constructed in polynomial time from Σ;
2. unrestricted answer containment is further reduced to unrestricted containment of $\text{CQ}^{\neg,\neq}$/$\text{UCQ}^{\neg,\neq}$ by *unfolding* P_Σ; unfolding means that whenever an atom B in the body of some rule r unifies with the head of some rule r', then B is replaced with $\mathsf{body}(r')$ (as in SLD resolution); the process is exhaustively repeated; if multiple rules $r_1, \ldots, r_n$ unify with B, then r is replaced with all n possible rewritings; since P_Σ is acyclic (because Σ is), the unfolding process terminates, however it may increase the size of P_Σ exponentially when some predicates are defined by multiple rules;
3. finally, if (the unfolding of) P_Σ is *positive* (i.e., it contains no negations nor any disequality), then unrestricted answer containment in the unfolded version of P_Σ is reduced to certain answering of CQs/UCQs against DL knowledge bases, see Theorem 2 below.

Theorem 2 says that there exists a PTIME reduction of unrestricted CQ (resp. UCQ) containment to the evaluation of certain answers of CQs (resp. UCQs) against DL knowledge bases.

Theorem 2. *Let Σ be a normalized specification and let P^U_Σ be the unfolding of P_Σ. For $i = 1, 2$, let Q_i be the definition of p_{S_i}, i.e. the set of rules $r \in P^U_\Sigma$ with p_{S_i} in $\mathsf{head}(r)$ (where S_1 and S_2 are the names of two graphs in Σ).*

If P_Σ is positive, then checking whether $p_{S_1} \subseteq_{KB,\Sigma} p_{S_2}$ can be reduced in polynomial time to evaluating for all $q \in Q_1$ an answer $\mathsf{c\_ans}(Q_2, KB_q)$, where KB_q is obtained from KB by binding the variables in q to fresh constants, and adding the instantiated body to KB's ABox as a set of assertions.

More precisely, for each $q \in Q_1$, one has to check whether the tuple of fresh constants assigned to the variables in $\mathsf{head}(q)$ belongs to $\mathsf{c\_ans}(Q_2, KB_q)$. Basically, the reduction is centred around a form of skolemization.

Note that this result is slightly different from the known relationships between query answering and query containment, since $p_{S_1} \subseteq_{KB,\Sigma} p_{S_2}$ is based on a nonstandard (unrestricted) notion of evaluation, similar to the one used for service comparison.

The above reduction suffices to derive complexity bounds for positive P_Σ. Note that P_Σ is positive when `else`, $\neq$, and $\neg$ are not supported, that is, in $\mathcal{SD}^u$ and its fragments. When Σ is formulated in $\mathcal{SD}^u$, then the unfolding of P_Σ may be exponentially larger, as the translation of unions into rules introduces predicates defined by multiple rules. It is not hard to see, however, that $\mathcal{SD}^{k,u}$ specifications lead to unfoldings that are only polynomially larger than P_Σ. Then the above reduction steps tell us that complexity of service comparison within $\mathcal{SD}^{k,u}$ and its fragments is bounded by the complexity of computing certain answers against DL KBs; for $\mathcal{SD}^u$ there is a further exponential explosion due to unfolding.

The complexity of query answering is NP-complete for $\mathcal{EL}$ [9], EXPTIME-complete for $\mathcal{DLR}$ [3], and co3NEXPTIME complete for $\mathcal{SHIQ}$ (cf. [6]). Moreover, query containment w.r.t. empty knowledge bases is NP-hard [4], and it is not difficult to see that the complexity of the standard reasoning tasks in $\mathcal{ALC}$ with general TBoxes (EXPTIME-complete) provides a lower bound to CQ answering against $\mathcal{ALC}$ KBs, so the upper bounds for $\mathcal{EL}$ and $\mathcal{ALC}$ are strict. These observations support the following theorem:

Theorem 3. *The complexity of service comparison in $\mathcal{SD}(\mathcal{X})$ and $\mathcal{SD}^{k,u}(\mathcal{X})$ is*

- *NP-complete for $\mathcal{X} = \mathcal{EL}$;*
- *EXPTIME-complete for $\mathcal{X}$ ranging from $\mathcal{ALC}$ to $\mathcal{DLR}$;*
- *in co3NEXPTIME for $\mathcal{X} = \mathcal{SHIQ}$.*

If the underlying description logic supports unrestricted negation (or equivalently, atomic negation and GCI), then negative literals in rule and query bodies (if any) can be *internalized* in the KB in a simple way: just replace each literal $\neg A(t)$ with $\bar{A}(t)$ where $\bar{A}$ is a fresh atom, and extend the TBox with the axioms $\bar{A} \sqsubseteq \neg A$ and $\neg A \sqsubseteq \bar{A}$. Internalization makes it possible to support constructs such as negated conditions, disequalities, and `else`, that introduces negation implicitly through the translations $[\bar{c}_i]$. After removing negation from P_Σ via internalization, we can exploit the available complexity results for the extensions of $\mathcal{ALC}$ (that allow internalization).

Theorem 4. *The complexity of service comparison in $\mathcal{SD}^\neg(\mathcal{X})$, and between $\mathcal{SD}^{k,e}(\mathcal{X})$ and $\mathcal{SD}^{k,u,e,\neg}(\mathcal{X})$ is*

- *in EXPTIME for $\mathcal{X} = \mathcal{EL}$;*

– *EXPTIME-complete for $\mathcal{X}$ ranging from $\mathcal{ALC}$ to $\mathcal{DLR}$;*
– *in co3NEXPTIME for $\mathcal{X}$ = $\mathcal{SHIQ}$.*

Remark 1. $\mathcal{EL}$ does not support negation, therefore internalization is not possible. In the above theorem we inherit the upper bound for $\mathcal{ALC}$, but whether this is a tight bound is still an open question.

From the above results, we derive upper complexity bounds for more general logics, without any nesting bounds. In the absence of nesting bounds and in the presence of disjunctive constructs like unions and conditionals, the unfolding of P_Σ may be exponential.

Theorem 5. *The complexity of service comparison in $\mathcal{SD}^{u,e,\neg}(\mathcal{X})$ is*

– *in 2-EXPTIME for $\mathcal{X}$ = $\mathcal{EL}$;*
– *in 2-EXPTIME for $\mathcal{X}$ ranging from $\mathcal{ALC}$ to $\mathcal{DLR}$;*
– *in 4-EXPTIME for $\mathcal{X}$ = $\mathcal{SHIQ}$.*

Also in this case, whether these bounds are tight is still an open question.

Currently, we do not know whether $\mathcal{SDL}_{full}$ or even its fragment $\mathcal{SD}^{\neq}$ are decidable. There exist some undecidability results for CQs and UCQs with disequalities, and we conjecture they can be carried over to service comparison. This will be a subject for future work.

	$\mathcal{EL}$	$\mathcal{ALC}$ $\mathcal{DLR}$	$\mathcal{SHIQ}$
$\mathcal{SD}$, $\mathcal{SD}^{k,u}$	NP-complete	EXPTIME-complete	in co3NEXPTIME
$\mathcal{SD}^{\neg}$	(in EXPTIME)	EXPTIME-complete	in co3NEXPTIME
$\mathcal{SD}^{k,e}$, $\mathcal{SD}^{k,e,\neg}$ $\mathcal{SD}^{k,u,e}$, $\mathcal{SD}^{k,u,e,\neg}$	(in EXPTIME)	EXPTIME-complete	in co3NEXPTIME
$\mathcal{SD}^{u}$, $\mathcal{SD}^{e}$, $\mathcal{SD}^{u,e}$ $\mathcal{SD}^{u,\neg}$, $\mathcal{SD}^{e,\neg}$, $\mathcal{SD}^{u,e,\neg}$	(in 2-EXPTIME)	in 2-EXPTIME	in 4-EXPTIME

Table 1. Some complexity results for decidable cases

7 Related work

The language introduced in [1], $\mathcal{SDL}(\mathcal{X})$, was based on an embedding of service comparison into subsumption in an expressive description logic, $\mu\mathcal{ALCIO}$. With the reduction to query containment we adopt here, it is possible to support service intersection and dataflow graphs, even if they violate the quasi-forest structure of $\mu\mathcal{ALCIO}$. Moreover, we provide an articulated complexity analysis not available in [1].

The idea of formalizing services as queries has been first introduced in [5]. The language adopted there is simpler than ours: only one construct combining our selection and restriction, and a form of composition where output and input messages must perfectly match. The semantics of services in [5] is restricted to the constants occurring in a

KB rather than domain elements. Furthermore, all upper bounds provided there are EXPTIME or beyond. Currently our NP bounds for $\mathcal{EL}$ identify the most efficient service description logics in the literature. Moreover, even in the hardest cases, our language is never more complex than [5].

In OWL-S, services are described by means of preconditions, postconditions, and add/delete lists. Pre- and postconditions are like ABoxes; add/delete lists specify the side effects of the services. The same mechanism can describe functional services. WSMO is built upon an articulated model, including user roles and goals, that lead to a planning-like view of service composition. In WSDL-S, WSDL service specifications (that are basically type definitions) are bound to concepts defined in an underlying ontology. No good computational results are currently available for any of the above standards.

8 Conclusions and future work

$\mathcal{SDL}_{full}$ and its fragments are rich service description languages that—however—enjoy numerous decidability results (reported in Table 1), and in some case ($\mathcal{SD}^{k,u}(\mathcal{EL})$) service comparison is significantly less complex than in previous competing logics. Encouraging experimental results are available for an analogous problem [2]. We are planning an experimental implementation based on the same technology.

Many issues need further work, here we mention just the main open problems. Automated service composition needs efficient heuristics for quickly selecting promising candidate dataflows. The bounds for $\mathcal{EL}$ reported in parentheses in Table 1 are simply inherited from more complex logic and it is not obvious whether they are tight. Disequalities and negation over roles would be helpful, but the undecidability results of [8] warn that some restrictions may be needed. It would also be interesting to check whether service comparison can be in NP also for other low-complexity logics such as the *DL-lite* family.

References

1. Piero A. Bonatti. Towards service description logics. In *JELIA, LNCS 2424*, pages 74–85. Springer, 2002.
2. Piero A. Bonatti and F. Mogavero. Comparing rule-based policies. In *9th IEEE Int. Work. on Policies for Distributed Systems and Networks (POLICY 2008)*, pages 11–18, 2008.
3. D. Calvanese, G. De Giacomo, and M. Lenzerini. Conjunctive query containment and answering under description logic constraints. *ACM Trans. Comput. Log.*, 9(3), 2008.
4. A.K. Chandra and P.M. Merlin. Optimal implementation of conjunctive queries in relational data bases. In *Proc. Ninth Annual ACM Symp. on Theory of Computing*, pages 77–90, 1976.
5. D. Hull, E. Zolin, A. Bovykin, I. Horrocks, U. Sattler, and R. Stevens. Deciding semantic matching of stateless services. In *Proc. of AAAI 2006*. AAAI Press, 2006.
6. Magdalena Ortiz, Diego Calvanese, and Thomas Eiter. Data complexity of query answering in expressive description logics via tableaux. *J. of Automated Reasoning*, 41(1):61–98, 2008.
7. Riccardo Rosati. Integrating ontologies and rules: Semantic and computational issues. In *Reasoning Web, LNCS 4126*, pages 128–151. Springer, 2006.
8. Riccardo Rosati. The limits of querying ontologies. In *ICDT, LNCS 4353*, pages 164–178. Springer, 2007.
9. Riccardo Rosati. On conjunctive query answering in $\mathcal{EL}$. In *Description Logics*. CEUR-WS.org, 2007.

Towards Large Scale Reasoning on the Semantic Web

Balázs Kádár, Gergely Lukácsy and Péter Szeredi

Budapest University of Technology and Economics
Department of Computer Science and Information Theory
1117 Budapest, Magyar tudósok körútja 2., Hungary
balazs@kadar.biz,{lukacsy,szeredi}@cs.bme.hu

Abstract. Traditional algorithms for description logic (DL) instance retrieval are inefficient for large amounts of underlying data. As description logic is becoming popular in areas such as the Semantic Web, it is very important to have systems that can reason efficiently over large data sets. In this paper we present the DLog description logic reasoner specifically designed for such scenarios.
The DLog approach transforms description logic axioms using the $\mathcal{SHIQ}$ DL language into a Prolog program. This transformation is done without any knowledge of the particular individuals: they are accessed dynamically during the normal Prolog execution of the generated program. This allows us to store the individuals in a database instead of memory, which results in better scalability and helps using description logic ontologies directly on top of existing information sources.
In this paper we focus on the description of the DLog application itself. We present the architecture of DLog and describe its interfaces. These make it possible to use ABoxes stored in databases and to communicate with the Protégé ontology editor, as a server application. We also evaluate the performance of the DLog database extension.
Keywords: large data sets, description logic, reasoning, logic programming, databases

1 Introduction

Description Logics (DLs) allow us to represent *knowledge bases* consisting of terminological axioms (the *TBox*) and assertional knowledge (the *ABox*).

Description Logics are becoming widespread as more and more systems start using semantics for various reasons. As an example, in the Semantic Web idea, DLs are intended to provide the mathematical background needed for more intelligent query answering. Here the knowledge is captured in the form of expressive ontologies, described in the Web Ontology Language (OWL) [1]. This language is mostly based on the $\mathcal{SHIQ}$ description logic, and it is intended to be the standard knowledge representation format of the Web.

However, we have tremendous amounts of information on the Web which calls for reasoners that are able to *efficiently* handle such abundance of data.

Moreover, as these data cannot be stored directly in memory, we need solutions for querying description logic concepts in an environment where the ABox is stored in a *database*.

We found that most existing description logic reasoners are not suitable for this task, as these are not capable of handling ABoxes stored externally. This is not a simple technical problem: most existing algorithms for querying DL concepts need to examine the whole ABox to answer a query. This results in scalability problems and undermines the point of using databases. Because of this, we started to investigate techniques which allow the separation of the inference algorithm from the data storage.

We have developed a solution, where the inference algorithm is divided into two phases. First we create a *query-plan* in Prolog from the actual DL knowledge base, without accessing the underlying data set. Subsequently, this query-plan can be run on real data, to obtain the required results. The implementation of these ideas is incorporated in the DLog reasoning system, available at `http://dlog-reasoner.sourceforge.net`.

In this paper we focus on the architecture of the DLog system, as well as on its external interfaces. We discuss the interface used for accessing databases, which allows description logic reasoning on top of existing information sources. We also describe the Protégé [2] interface that makes it possible to use DLog as the back-end reasoner of this popular ontology editor. Details on the theoretical side of DLog can be found in [3] and in [4].

This paper is structured as follows. Section 2 summarises related work. In Section 3 we give a general introduction to the DLog approach and present the architecture and implementation details of the system. The database and Protégé interfaces are described in Sections 4 and 5, respectively. Section 6 evaluates the performance of the database extension of DLog w.r.t. the version which stores the ABox as Prolog facts. Finally, in Section 7, we conclude with the future work and the summary of our results.

2 Related work

Several techniques have emerged for dealing with ABox-reasoning. Traditional ABox-reasoning is based on the *tableau inference* algorithm, which tries to build a model showing that a given concept is satisfiable. To infer that an individual i is an instance of a concept C, an indirect assumption $\neg C(i)$ is added to the ABox, and the tableau-algorithm is applied. If this reports inconsistency, i is proved to be an instance of C. The main drawback of this approach is that it cannot be directly used for high volume instance retrieval, because it would require checking all instances in the ABox, one by one.

To make tableau-based reasoning more efficient on large data sets, several techniques have been developed in recent years [5]. These are used by the state-of-the-art DL reasoners, such as RacerPro [6] or Pellet [7].

Extreme cases involve serious restrictions on the knowledge base to ensure efficient execution with large amounts of instances. For example, [8] suggests a

solution called the *instance store*, where the ABox is stored externally, and is accessed in a very efficient way. The drawback is that the ABox may contain only axioms of form $C(a)$, i.e. we cannot make role assertions.

Paper [9] discusses how a first order theorem prover such as Vampire can be modified and optimised for reasoning over description logic knowledge bases. This work, however, mostly focuses on TBox reasoning.

In [10], a resolution-based inference algorithm is described, which is not as sensitive to the increase of the ABox size as the tableau-based methods. However, this approach still requires the input of the *whole content* of the ABox before attempting to answer any queries. The KAON2 system [11] implements this method and provides reasoning services over the description logic language $\mathcal{SHIQ}$ by transforming the knowledge base into a disjunctive datalog program.

Although the motivation and goals of KAON2 are similar to ours, unlike KAON2 (1) we use a pure two-phase reasoning approach (i.e. the ABox is accessed only during query answering) and (2) we translate into Prolog which has well-established, efficient and robust implementations.

Article [12] introduces the term Description Logic Programming. This idea uses a direct transformation of $\mathcal{ALC}$ description logic concepts into definite Horn-clauses, and poses some restrictions on the form of the knowledge base, which disallow axioms requiring disjunctive reasoning. As an extension, [13] introduces a fragment of the $\mathcal{SHIQ}$ language that can be transformed into Horn-clauses. This work, however, still poses restrictions on the use of disjunctions.

3 The DLog system

The main idea of the DLog approach is that we transform a $\mathcal{SHIQ}$ knowledge base KB into first-order clauses $\Omega(KB)$ and from these we generate Prolog code [3]. In contrast with [11], all clauses containing function symbols are eliminated during the transformation: the resulting clauses can be resolved further only with ABox clauses. This forms the basis of a pure two phase reasoning framework, where every possible ABox-independent reasoning step is performed before accessing the ABox itself, allowing us to store the content of the ABox in an external database.

Actually, in the general transformation, we use only certain properties of $\Omega(KB)$. These properties are satisfied by a subset of first order clauses that is, in fact, larger than the set of clauses that can be generated from a $\mathcal{SHIQ}$ KB. We call these clauses *DL clauses*. As a consequence of this, our results can be used for DL knowledge bases that are more expressive than $\mathcal{SHIQ}$. This includes the use of certain role constructors, such as union. Furthermore, some parts of the knowledge base can be supplied by the user directly in the form of first order clauses. More details can be found in [3].

As the clauses of a $\mathcal{SHIQ}$ knowledge base KB are normal first-order clauses we can apply the Prolog Technology Theorem Proving (PTTP) technology [14] directly on these. In [3] we have simplified the PTTP techniques for the special

case of DL clauses and we have proved that these modifications are sound and complete for DL clauses.

The simplified PTTP techniques used in DLog include deterministic *ancestor resolution* and *loop elimination*. Both are applicable only to unary predicates, i.e. predicates corresponding to DL concepts.

In the design of the DLog system we focus on modularity. This enables us to easily implement new features and new interfaces. The top level architecture of the system is shown in Figure 1. In this figure, as in subsequent figures of the paper, rectangles with rounded corners represent modules of the DLog system, while data are shown as plain rectangles. In Figure 1 the DLog reasoner is shown within a dashed rectangle.

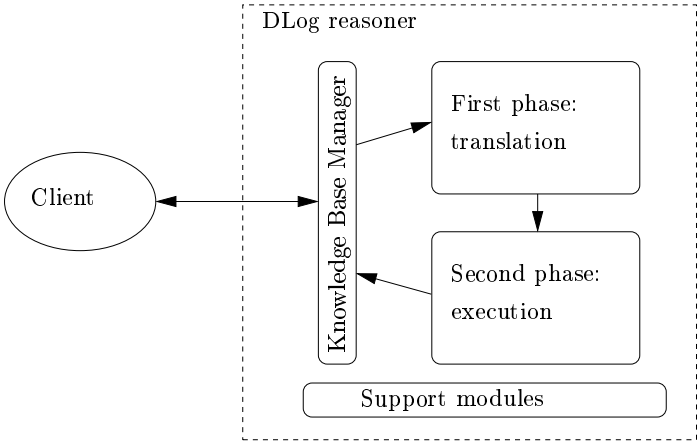


Fig. 1. The top level architecture of the DLog system.

The user (either local or remote) accesses DLog through one of the external interfaces. These interfaces range from a local console to server interfaces like DIG used by the Protégé ontology editor. The knowledge base manager is the central piece of the system. It coordinates the tasks of the other modules, and performs the administration of multiple concurrent knowledge bases. It forwards the request arriving from the interfaces to the reasoner modules.

The *support modules* consist of several tools that are used by most parts of the system. They include a configuration manager module, a logger, an XML reader, a run-time system for the second phase, and several portability tools that allow DLog to run under different Prolog implementations (currently SWI and SICStus).

The first phase, translation, shown in Figure 2, takes a set of description logic axioms as input. These axioms are divided into two parts: the TBox or terminology box stores concept and role inclusion axioms, while the ABox or assertion box contains the factual data. The ABox may be stored (partly or

completely) in external databases. The ABox is processed first, producing the *ABox code* (which is a Prolog module), and the ABox signature, which is required for translating the TBox. The generation of ABox code includes optimisations such as indexing on second argument for roles stored in memory.

Next, the TBox is processed in two steps. First the DL translator module transforms the description logic formulae to a set of DL clauses [15], which are passed on to the TBox translator module that generates the executable *TBox code*. This generated code is equivalent, with respect to instance retrieval, to the input DL knowledge base. The TBox translator module uses various optimisations [3] to obtain more efficient Prolog programs. The ABox and TBox code can be generated directly into memory or may be saved to disk for later (standalone) use.

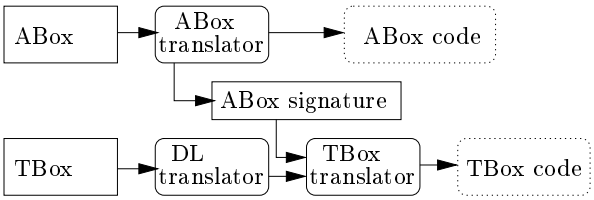


Fig. 2. The first phase: translation.

The second phase, execution, shown in Figure 3, uses the ABox and TBox programs generated in the first phase, to answer queries. There are two ways to execute queries: the generated TBox can be called directly from Prolog as a low-level interface, or the *Query module* provides a high-level interface that provides basic support for composite queries and can aggregate the results. In normal operation the query module is called by the knowledge base manager, which forwards the results to the user interface. As the query module does not depend on the rest of the system, it may be used in standalone operation. The run-time system (shown as RTS in the figure) includes a hash table implemented in C used to speed up the reasoning, and optional collection of statistics.

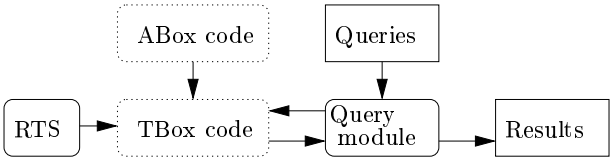


Fig. 3. The second phase: execution.

4 Integrating DLog with databases

As the first phase of reasoning (i.e. the generation of a query plan) only depends on the signature of the data set, and because of the top-down inference of Prolog, DLog can efficiently use databases to store the ABox.

There may be several advantages in using databases to store the ABox. Firstly, this allows reasoning on data sets that cannot fit into memory. Secondly, it makes integrating DLog with existing systems easier, as the reasoner can use the existing databases of other applications. Thirdly, querying some concepts (namely those corresponding to so-called *query predicates*) may be performed using complex database queries, rather than DL reasoning, which is expected to deliver a marked increase in performance.

A predicate is a *query predicate* [3], if it is non-recursive, it does not invoke its negation, and is not invoked from within its negation. Here, a predicate P_0 is said to invoke a predicate P_n, $n \geq 1$, if there are $n-1$ intermediate predicates $P_1 \dots P_{n-1}$, such that P_i is directly invoked by P_{i-1}, i.e. it occurs in a clause body the head of which is P_{i-1}, for $i = 1, \dots, n$.

Query predicates require neither loop elimination, nor ancestor resolution during execution. The name "query predicate" reflects that fact that such predicates can be transformed to complex database queries (provided that all concepts and roles required are stored in a single database). This can increase the performance as the database engine can optimise the query using statistical and structural knowledge of the database in question.

We designed the database interface to be as simple as possible. The databases are accessed via the ODBC driver of SWI-Prolog; as a consequence DLog can interface with most modern database systems. We wanted a way to specify database access using existing tools and interfaces – such as Protégé and the DIG interface it utilises – even if those do not, at the moment, provide a way to specify database usage. To access a database, several pieces of information are needed: the name of the database, a user name, a password, a description of which table to use for given concepts and roles, etc. Because of the aforementioned requirements we decided to use ABox assertions to carry this meta-information. ABox assertions are description logic constructs that are readily available in DL systems and interfaces, such as OWL and DIG.

In order to specify the database access for concepts and roles we introduce new roles (object properties), attributes (datatype properties) and individuals defined in the namespace `http://www.cs.bme.hu/dlogDB`.

The ODBC interface prescribes that database connections are to be identified by a *Data Source Name* (DSN). In DLog we introduce an individual to represent a given database connection. Roles and concepts are also represented by individuals. An arbitrary name can be used for such an individual.

The meta data provided is used to connect to the database, and, for each concept and role, an additional clause is generated, which, by executing an appropriate database query, lists appropriate individuals (or pairs of individuals). This allows concepts and roles to be stored partially in databases and partially in memory. This may be very useful when developing ontologies.

4.1 Specifying the Database Interface

Database connections are represented by individuals that have the string attribute `hasDSN` defined. The value of this attribute is the name of the data source (DSN). As all other names in this section, this name is defined in the namespace `http://www.cs.bme.hu/dlogDB`. Additional string attributes, namely `hasUserName` and `hasPassword`, may be used to specify the user name and the password for the given connection, if required.

The object property `hasConnection` links an individual representing a role or a concept with the database connection to be used for accessing it. This makes it possible to use one data source for one concept, and a different one for another. The instance on the left hand side is the individual representing the role or concept, while the instance on the right hand side is the individual representing the connection.

Two methods are provided to specify how to get the data from the database. One is to specify a query that is to be directly executed on the database. This method, named the *simple interface*, is provided because of its simplicity: it can be applied to databases without any modification. However it has two drawbacks:

- it makes transforming query predicates to database queries very difficult; and
- it performs badly for instance check queries.

The latter is a large setback as most of the queries are instance checks, assuming the the *projection* optimisation of [3] is used.

Therefore the second, preferred, way is to provide the name of a table or of a view and the name of the column(s) of this table. This approach, called the *complex interface* may require the creation of new views in the database, but provides much greater flexibility and better performance.

The SQL query in the simple interface is defined using the string attribute `hasQuery`. The individual represents the role or concept and the attribute value is the query string. For individuals representing roles the query must return two columns, and for those used for concepts it must return one column that contains the individual name.

If the complex interface is used, the name of the table or view to use is specified by the string attribute `hasTable`. The name of the column listing the individuals of a concept is given using the string attribute `hasColumn`. For roles, the attributes `hasLHS` and `hasRHS` are used for the left and the right hand side, respectively.

Because, in Protégé, individuals cannot be specified as instances of a negated concept, we provide some additional attributes: `hasNegQuery`, `hasNegTable` and `hasNegColumn`. These are used to specify the database access of negated concepts, in a way similar to their respective positive pairs. By providing an attribute `hasNegQuery` for a name representing the concept C we specify a query listing the individuals of $\neg C$. Obviously, both `hasQuery` and `hasNegQuery` can appear as attributes of the same individual.

To specify that the individual `concept` represents the concept C, one simply has to make `concept` an instance of C. The DLog system will check each concept occurring in the ABox if it contains an instance which is in the namespace `http://www.cs.bme.hu/dlogDB`. If such an instance is found, it is interpreted as a "handle" to a database which is to produce (additional) instances for the given concept.

Similarly, to specify that an individual `role` represents the role R, we require that the user includes the triple `{role, R, indiv}` in the ABox. Here `indiv` is an arbitrary individual. Again DLog will look for an instance in the namespace `http://www.cs.bme.hu/dlogDB` within the domain (i.e. the left hand side) of each role, and use it to construct a database access for the given role.

The database interface is currently in the alpha test phase. We believe that our approach for this task, discussed above, is an intermediate solution. Ultimately the standard interfaces, such as DIG, should be extended to allow storing (parts of) the ABox in databases. However, we hope that our work contributes to implementing this ultimate goal.

4.2 Examples of Using the Database Interface

We now present two examples for interfacing with databases, one for the simple, and one for the complex interface.

The examples contain ABox assertions, which are displayed as RDF triples in `{subject, predicate, object}` format. String values are shown between quotes. The namespace `http://www.cs.bme.hu/dlogDB#` is represented by the `dlog:` prefix.

Figure 4 shows the use of the simplified interface for the ABox of the *Iocaste* example. This classical example involves the concept describing a person having a patricide child, who, in turn, has a non-patricide child. The ABox axioms, which are now to be stored in a database, describe the `hasChild` relation between pairs of individuals (traditionally containing `(Iocaste, Oedipus)`, `(Iocaste, Polyneikes)`, `(Oedipus, Polyneikes)` and `(Polyneikes, Thersandros)`). The ABox also specifies which individuals are patricide and which are non-patricide (traditionally `Oedipus` is known to belong to the former, while `Thersandros` to the latter).

We have chosen the namespace represented by the `io:` prefix for the names in this ontology. The database connection is named `iodb`, and the corresponding DSN is specified as `"iocaste"` (line 1). This connection is accessed without specifying a user name or a password. Accordingly, `iodb` has no attributes other than `dlog:hasDSN`.

Both the role `hasChild` and the concept `Patricide` are taken from this database. The role `hasChild` is represented by the instance `dlog:riohasChild`. We chose this name as a mnemonic for a *r*ole from the namespace *io*, called *hasChild*, but any other name could have been used. Line 2 tells the system that this individual represents the role `io:hasChild`. Here, the right hand side of the role is of no interest, so we chose to have the same individual as on the left hand side. Line 6 tells that the individual `dlog:cioPatricide` is an instance of

```
1  {dlog:iodb, dlog:hasDSN, "iocaste"}
2  {dlog:riohasChild, io:hasChild, dlog:riohasChild}
3  {dlog:riohasChild, dlog:hasConnection, dlog:iodb}
4  {dlog:riohasChild, dlog:hasQuery,
5                       "SELECT parent, child FROM hasChild"}
6  {dlog:cioPatricide, rdf:type, io:Patricide}
7  {dlog:cioPatricide, dlog:hasConnection, dlog:iodb}
8  {dlog:cioPatricide, dlog:hasQuery,
9                        "SELECT name FROM people WHERE patricide"}
10 {dlog:cioPatricide, dlog:hasNegQuery,
11                       "SELECT name FROM people WHERE NOT patricide"}
```

Fig. 4. An example of the simplified database interface.

the concept `io:Patricide`[1]. This individual, which thus represents the concept `io:Patricide`, has two queries associated with it: one for `io:Patricide` (line 8) and one for its negation (line 10).

The simplified interface allows complex queries, such as the one for Patricide which has a `WHERE` clause. This way the existing table `people` can be used without modification. However, this approach makes it very difficult to transform any possible query predicates in the TBox to direct database queries, and instance check queries run with a poor performance.

We now present a second example. The TBox of this example, taken from [4], is shown below.

```
1  ∃hasFriend.Alcoholic  ⊑  ¬Alcoholic
2  ∃hasParent.¬Alcoholic  ⊑  ¬Alcoholic
```

Line 1 describes that those who have a friend who is alcoholic are non-alcoholic (as they see a bad example), while line 2 states that those who have a non-alcoholic parent are non-alcoholic (as they see a good example). In the classic form the ABox contains role assertions for the `hasParent` and `hasFriend` relations only, and no concept assertions about anyone being alcoholic or non-alcoholic. In spite of this, in the presence of certain role instance patterns, one can infer some people to be non-alcoholic, using case analysis.

For example, consider the following pattern: Jack is Joe's parent and also his friend. Now, if we assume that Jack is alcoholic, then the axiom in line 1 implies that Joe is not alcoholic. On the other hand, if Jack is not alcoholic, it follows from line 2 that Joe is not alcoholic, either. Thus these two role assertions imply that Joe has to be non-alcoholic. Other patterns, where Joe can be inferred to be non-alcoholic, are the following: Joe is a friend of himself; Joe is a friend of an ancestor; and Joe's two ancestors are in the `hasFriend` relationship.

[1] Note that the prefix `rdf`, used in the predicate position of the triple in line 6, refers to the RDF namespace: `http://www.w3.org/1999/02/22-rdf-syntax-ns#`.

In Figure 5 we present a database access specification for the above example, using the complex interface. Here, the database `alcoholic` is accessed with the user name `"drunkard"` and the password `"palinka"` (lines 1–3). We assume that a new view, called `"hasParentView"`, was defined in the database to hide the complex query for the role `hasParent`, cf. lines 4–6. The columns of this view, `child` and `parent` (lines 7–8), contain the data for the role `hasParent`. From this information DLog can create a query for instance retrieval (`"SELECT child, parent FROM hasParentView"`), and three other query patterns for the cases when at least one of the individuals is known (e.g. `"SELECT child FROM hasParentView WHERE parent = ?"`). This approach allows for the generation of complex database queries for the query predicates.

```
1  {dlog:alcdb, dlog:hasDSN, "alcoholic"}
2  {dlog:alcdb, dlog:hasUserName, "drunkard"}
3  {dlog:alcdb, dlog:hasPassword, "palinka"}
4  {dlog:ralchasParent, alc:hasParent, dlog:ralchasParent}
5  {dlog:ralchasParent, dlog:hasConnection, dlog:alcdb}
6  {dlog:ralchasParent, dlog:hasTable, "hasParentView"}
7  {dlog:ralchasParent, dlog:hasLHS, "child"}
8  {dlog:ralchasParent, dlog:hasRHS, "parent"}
9  {dlog:ralchasFriend, alc:hasFriend, dlog:ralchasFriend}
10 {dlog:ralchasFriend, dlog:hasConnection, dlog:alcdb}
11 {dlog:ralchasFriend, dlog:hasTable, "friends"}
12 {dlog:ralchasFriend, dlog:hasLHS, "friend1"}
13 {dlog:ralchasFriend, dlog:hasRHS, "friend2"}
14 {dlog:calcAlcoholic, rdf:type, alc:Alcoholic}
15 {dlog:calcAlcoholic, dlog:hasConnection, dlog:alcdb}
16 {dlog:calcAlcoholic, dlog:hasTable, "alcoholicView"}
17 {dlog:calcAlcoholic, dlog:hasColumn, "name"}
18 {dlog:calcAlcoholic, dlog:hasNegTable, "nonalcoholicView"}
19 {dlog:calcAlcoholic, dlog:hasNegColumn, "name"}
```

Fig. 5. An example of the complex database interface.

In Figure 5, lines 10–13 specify the database access for the role `hasFriend`, while lines 14–19 allow for accessing individuals belonging to the concept `alcoholic` and its negation through appropriate database views.

5 Integrating DLog with Protégé

Protégé [2] is an open source ontology editor that supports the Web Ontology Language (OWL) [1], and can connect to reasoners via the HTTP-based DIG interface [16]. The DLog server implements the DIG interface and can be used to execute instance retrieval queries issued from the graphical interface of Protégé.

The DIG interface specifies communication via HTTP, and uses XML data format. For the implementation we used the HTTP server provided with SWI-Prolog. In implementing the interface we faced difficulties caused by some ambiguities of the DIG specifications, despite there being an (exact) XML schema definition. Another difficulty was that Protégé does not strictly follow the definition of the interface. For example it uses a `clearKB` command that is not even defined in version 1.1 of DIG. In DIG 1.0, which supported only a single database, this command was defined, but Protégé uses the new version that supports multiple concurrent knowledge bases. We strove for an implementation as generic and complying to the interface definition as possible while, also being compatible with Protégé.

For parsing XML we use the SGML module of SWI-Prolog, which can be operated in an XML compatibility mode, allowing namespaces. As this is not a direct XML parser, it has some difficulties when used in XML mode. For example even with the strictest settings and treating all warnings as errors, it accepts input files that are not even well-formed XML. Because of this, and in hope of better performance, we are planning to switch to Apache Xerces-C++. With Xerces we plan to use SAX parsing, instead of DOM, with the hope of lower memory usage and faster parsing.

The data are extracted from the XML DOM using Definite Clause Grammars (DCG).

Figure 6 shows the results of a query issued from Protégé, as answered by the DLog server.

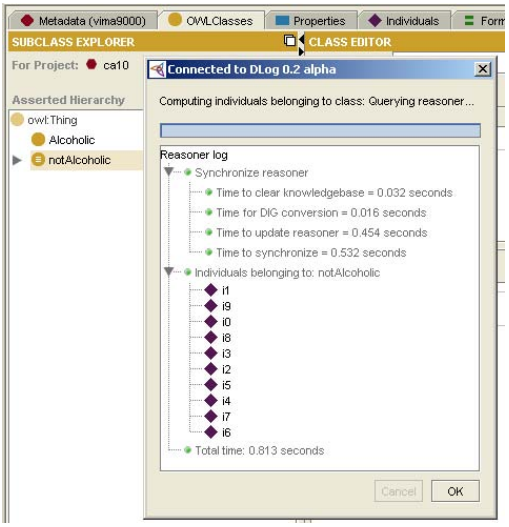


Fig. 6. Screenshot of query results in Protégé answered by DLog.

The integration of Protégé and the database interface is in progress. A serious difficulty is that if the results of a query contain individuals that are not defined in Protégé (i.e. individuals present only in databases) Protégé silently drops these individuals from the list of query results.

6 Evaluation

This section contains a preliminary performance test of the database interface.

We tried the database interface on a large version of the Iocaste problem which contains 5058 pairs in the `hasChild` relation, 855 instances that are known to be patricide, and 314 that are known to be non-patricide.

The execution results are summarised in Table 1. The load time means the time it takes to load the file which contains the axioms, including the XML parsing. The translation time is the time it takes to generate the TBox and ABox code from the axioms, while execution time is the run-time of the query.

Table 1. Comparing the in-memory and database version of a large Iocaste test.

(seconds)	load	translate	execute	total
in-memory	0.88	0.53	0.02	1.43
database	0.05	0.02	0.36	0.43

When the ABox is stored in memory, the translation takes 1.41 seconds, and the execution takes only 0.02 seconds. Note that these figures were obtained with the indexing optimisation turned off. When this optimisation is turned on, the number of generated ABox clauses is doubled, and translation time increases accordingly.

The database variant of the example enumerates all the instances of the queried concept in 0.36 seconds. This, compared to the original 0.02 seconds is much slower. However, the time we spent at compile-time was altogether 0.07 seconds, resulting in a total execution time of 0.43 seconds. To sum up, in terms of total query execution time, more than a three-fold decrease was achieved, using the database interface.

From the above data it may seem that using a database for storing the ABox, which fits into memory, is beneficial only because of the reduced compile-time. However, we believe that in the case of large data sets and complex queries (especially if these contain concepts giving rise to query predicates) execution time can also be better than that of the in-memory variant.

Detailed evaluation of the DLog System can be found in [3].

7 Summary and future work

In this paper we have shown the architecture of the DLog system, discussed a database interface for representing large ABoxes, and reported on the integration of DLog with the Protégé ontology editor.

The database interface is especially useful if the data set cannot fit in memory or if it is shared with other systems. Using databases can greatly reduce compile time and, with advanced optimisations, it may provide efficiency similar to that of the in-memory version.

Future improvements include the optimisation of query predicates, by transforming them to database queries, and better integration of Protégé and the database interface. Our plans also include the implementation of a query module to handle composite queries, and the support for additional interface formats, such as OWL, or the KRSS notation used by e.g. the RacerPro engine.

Acknowledgements

The authors are grateful to the anonymous reviewers for their comments on the earlier version of the paper, and especially for recommending the *Billion Triples Challenge* for evaluation.

References

1. Bechhofer, S.: OWL web ontology language reference. W3C recommendation (February 2004)
2. Noy, N., Fergerson, R., Musen, M.: The knowledge model of Protege-2000: Combining interoperability and flexibility. http://citeseer.nj.nec.com/noy01knowledge.html (2000)
3. Lukácsy, G., Szeredi, P.: Efficient description logic reasoning in Prolog: the DLog system. Technical report, Budapest University of Technology and Economics (January 2008) Conditionally accepted for publication in Theory and Practice of Logic Programming.
4. Lukácsy, G., Szeredi, P., Kádár, B.: Prolog based description logic reasoning. (December 2008) To appear in ICLP 2008.
5. Haarslev, V., Möller, R.: Optimization techniques for retrieving resources described in OWL/RDF documents: First results. In: Ninth International Conference on the Principles of Knowledge Representation and Reasoning, KR 2004, Whistler, BC, Canada, June 2-5. (2004) 163–173
6. Haarslev, V., Möller, R., van der Straeten, R., Wessel, M.: Extended Query Facilities for Racer and an Application to Software-Engineering Problems. In: Proceedings of the 2004 International Workshop on Description Logics (DL-2004), Whistler, BC, Canada, June 6-8. (2004) 148–157
7. Sirin, E., Parsia, B., Grau, B.C., Kalyanpur, A., Katz, Y.: Pellet: A practical OWL-DL reasoner. Web Semant. **5**(2) (2007) 51–53
8. Horrocks, I., Li, L., Turi, D., Bechhofer, S.: The Instance Store: DL reasoning with large numbers of individuals. In: Proceedings of DL2004, British Columbia, Canada. (2004)

9. Horrocks, I., Voronkov, A.: Reasoning support for expressive ontology languages using a theorem prover. In: FoIKS. Volume 3861 of Lecture Notes in Computer Science., Springer (2006) 201–218
10. Hustadt, U., Motik, B., Sattler, U.: Reasoning for Description Logics around SHIQ in a resolution framework. Technical report, FZI, Karlsruhe (2004)
11. Motik, B.: Reasoning in Description Logics using Resolution and Deductive Databases. PhD thesis, Univesität Karlsruhe (TH), Karlsruhe, Germany (January 2006)
12. Grosof, B.N., Horrocks, I., Volz, R., Decker, S.: Description logic programs: Combining logic programs with description logic. In: Proc. of the Twelfth International World Wide Web Conference (WWW 2003), ACM (2003) 48–57
13. Hustadt, U., Motik, B., Sattler, U.: Data complexity of reasoning in very expressive description logics. In: Proceedings of the Nineteenth International Joint Conference on Artificial Intelligence (IJCAI 2005), International Joint Conferences on Artificial Intelligence (2005) 466–471
14. Stickel, M.E.: A Prolog technology theorem prover: a new exposition and implementation in Prolog. Theoretical Computer Science **104**(1) (1992) 109–128
15. Zombori, Zs.: Efficient two-phase data reasoning for description logics. In: Proceedings of the International Federation for Information Processing Technical Committee on Artificial Intelligence (TC12), Milan, Italy (September 2008) Accepted conference paper.
16. Bechhofer, S.: The DIG description logic interface. http://dig.cs.manchester.ac.uk/ (2006)

Reasoning on the Web with Open and Closed Predicates

Gerd Wagner[1], Adrian Giurca[1], Ion-Mircea Diaconescu[1], Grigoris Antoniou[2], Anastasia Analyti[2] and Carlos Viegas Damasio[3]

[1] Brandenburg University of Technology, Germany
{G.Wagner, Giurca, M.Diaconescu}@tu-cottbus.de,
[2] Institute of Computer Science, FORTH-ICS, Greece
{antoniou, analyti}@ics.forth.gr
[3] Universidade Nova de Lisboa, Portugal
cd@di.fct.unl.pt

Abstract. SQL, Prolog, RDF and OWL are among the most prominent and most widely used computational logic languages. However, SQL, Prolog and RDF do not allow the representation of negative information, only OWL does so. RDF does not even include any negation concept. While SQL and Prolog only support reasoning with closed predicates based on negation-as-failure, OWL supports reasoning with open predicates based on classical negation, only. However, in many practical application contexts, one rather needs support for reasoning with both open and closed predicates. To support this claim, we show that the well-known Web vocabulary *FOAF* includes all three kinds of predicates i.e. *closed*, *open* and *partial* predicates. Therefore, reasoning with FOAF data, as a typical example of reasoning on the Web, requires a formalism that supports the distinction between open and closed predicates. We argue that *ERDF*, an extension of RDF, offers a solution to deal with this problem.

1 Introduction

1.1 Open and Closed Predicates

Many information management scenarios deal with predicates with their complete extension recorded (e.g. in a database). Such *closed* predicates use the computational mechanism of *negation-as-failure (NAF)* in order to infer negative conclusions based on the explicit absence (or non-inferability) of an information item. In other words, not for *open* but only for *closed* predicates, NAF is similar with standard negation.

The issue of reasoning with closed predicates and NAF has been researched in the field of Artificial Intelligence back in the 1980's, as a form of NAF has been implemented at that time both in the database language SQL and in the logic programming language Prolog. The resulting theories and formalisms, including the famous "Closed-World Assumption", have considered NAF to be the negation concept of choice in computational logic systems, and have downplayed the

significance of "open-world" reasoning with classical negation. 20 years later, however, a computational logic concept of classical negation has been chosen and implemented in a prominent computational logic formalism, viz the Web ontology language OWL [10]. While SQL and Prolog have a nonmonotonic computational logic semantics and support only closed predicates, OWL is based on a computational fragment of classical logic and therefore supports only open predicates. However, in many practical application contexts, one rather needs support for reasoning with both open and closed predicates.

1.2 Total and Partial Predicates

In fact, in addition to the distinction between *open* and *closed* predicates, it is useful to make another distinction between *total* and *partial* predicates. All these distinctions are related to the semantics of negative information and negation. The distinction between total and partial predicates is supported by *partial logic* (see [9]), which comes in different versions (with either 3 or 4 truth values) and can be viewed as a refinement of classical logic allowing both truth value gaps and truth value clashes. The law of the excluded middle only holds for total, but not for partial predicates. Both closed and open predicates are total. Consequently we obtain three kinds of predicates, as described in the following table:

	NAF=NEG	LEM (Law of Excluded Middle)
closed	yes	yes
open	no	yes
partial	no	no

The symbolic equation *NAF=NEG* denotes the condition that negation-as-failure and standard negation collapse, i.e. that both connectives are logically equivalent.

1.3 Three Kinds of Predicates in FOAF

A well-known example of a Web vocabulary is FOAF, the *Friends of a Friend* vocabulary [6], which is essentially expressed in RDFS (with a few additional constructs borrowed from OWL), and which has the purpose to create a Web of machine-readable information describing people, the links between them and the things they create and do. As examples of closed, open and partial predicates included in FOAF we consider the properties `foaf:member`, `foaf:knows` and `foaf:topic_interest`. Of course, one could simply stipulate that these predicates have a standard classical logic semantics. But we argue that their intended meaning in natural language implies that they are better treated as closed, open, respectively partial predicates according to partial logic.

When a `foaf:Group` is defined, we may assume that such a definition is not made in an uncontrolled distributed manner, but rather in a controlled way where one specific person (or agent) has the authority to define the group, typically in the context of an organization that empowers the agent to do so.

In this case, it is natural to consider the definition of the group membership to be a complete specification, and, consequently, to consider the `foaf:member` property to be a closed predicate. For the following example,

```
<foaf:Group rdf:ID="http://tu-cottbus.de/lit/erdf-team">
  <foaf:name>BTU Cottbus ERDF Team</foaf:name>
  <foaf:member rdf:resource="#Gerd"/>
  <foaf:member rdf:resource="#Adrian"/>
  <foaf:member rdf:resource="#Mircea"/>
</foaf:Group>
<foaf:Person rdf:ID="Gerd">
<foaf:Person rdf:ID="Adrian">
<foaf:Person rdf:ID="Mircea">
```

this would mean that we can draw the (negative) conclusion that

> Grigoris is not a member of the BTU Cottbus ERDF Team

based on the absence of a fact statement that "Grigoris is a member of the BTU Cottbus ERDF Team".

In the case of the property `foaf:knows`, however, we could argue that the standard RDF and OWL treatment of classes and properties as open predicates is adequate, since one does normally not make a complete set of statements about all persons one knows in a FOAF file. Consequently, the absence of a fact statement that "Grigoris knows Gerd" does not justify to draw the negative conclusion that "Grigoris does not know Gerd".

Both `foaf:member` and `foaf:knows` can be considered as *total* predicates that are subject to the *law of the excluded middle*, implying that the following disjunctive statements hold:

> Either Grigoris is a member of the BTU Cottbus ERDF Team or Grigoris is not a member of the BTU Cottbus ERDF Team.
> Either Grigoris knows Gerd or Grigoris does not know Gerd.

In the case of the property `foaf:topic_interest`, the situation is different. First, notice that while in the previous cases of `foaf:member` and `foaf:knows` there is no need of representing negative fact statements, we would like to be able to express both topics in which we are interested and topics in which we are definitely not interested (and would therefore prefer not to receive any news messages related to them). For instance, we may want to express the negative triple "Gerd is definitely not interested in the topic *motor sports*". Therefore, we should declare `foaf:topic_interest` to be a *partial* property, which means (1) that we want to be able to represent negative fact statements along with positive fact statements involving this predicate and (2) that the *law of the excluded middle* does not hold for it: it is not the case that for any topic x,

> Gerd is interested in the topic x or Gerd is (definitely) not interested in the topic x.

There may be topics, for which it is undetermined whether Gerd is interested in them or not.

1.4 Extended RDF

Since RDF(S) (see [7, 5, 3]) does not allow to represent negative information and does not support any negation concept, we need to extend it for turning it into a suitable reasoning formalism for FOAF and similar Web vocabularies.

In [12], it was argued that a database, as a knowledge representation system, needs two kinds of negation, namely *weak negation* for expressing *negation-as-failure* (or *non-truth*), and *strong negation* for expressing *explicit negative information* or *falsity*, to be able to deal with partial information. In [13], this point was also made for the Semantic Web as a framework for knowledge representation in general, and in [1, 2] for the Semantic Web language RDF with a proposal how to extend RDF for accommodating the two negations of partial logic as well as derivation rules. The extended language, called *Extended RDF*, or in short *ERDF*, has a model-theoretic semantics that is based on partial logic [9].

1.5 Plan of the Paper

While the theoretical foundation of ERDF has been presented in [1, 2], the novel contributions of this paper are

1. an exposition and discussion of the RDF-style syntax of ERDF, and
2. a presentation of a case study that shows how a practical Web vocabulary (FOAF) would benefit from the extended logical features offered by ERDF (the support of two kinds of negation and three kinds of predicates).
3. a discussion about our current implementation of ERDF tool set, including an inference engine, and our future plans for improvements.

2 The ERDF Abstract Syntax

This section describes the abstract syntax of ERDF in terms of a MOF/UML metamodel that is aligned with the RDF metamodel of OMG's *Ontology Definition Metamodel (ODM)* [8].

2.1 The ERDF-Vocabulary

ERDF adds the following classes to the RDFS vocabulary: `erdf:PartialClass`, `erdf:PartialProperty`, `erdf:TotalClass`, `erdf:TotalProperty`, `erdf:OpenClass`, `erdf:OpenProperty`, `erdf:ClosedClass` and `erdf:ClosedProperty`. These classes specialize `erdf:Class` and `erdf:Property` as depicted in Figure 1.

ERDF allows to designate properties and classes that are completely represented in a knowledge base – they are called *closed*. The classification if a predicate is closed or not is up to the owner of the knowledge base: the owner must know for which predicates there is complete information and for which there is not.

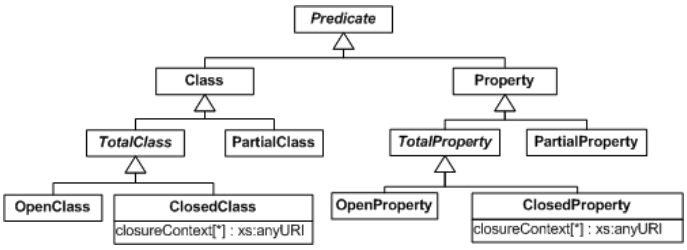


Fig. 1. The ERDF vocabulary as an extension of the RDFS vocabulary

2.2 ERDF Descriptions and Atoms

ERDF descriptions, as depicted in the metamodel diagram in Figure 2, extend RDF descriptions by

1. adding to RDF property-value slots an optional attribute `negationMode` that allows to specify three kinds of negation (`Naf` for *negation-as-failure*, `Sneg` for *strong negation* and `NafSneg` for *negation-of-failure over strong negation*). An optional value, `None` is also possible and it is the default value (i.e. when the attribute *negationMode* is missing);
2. allowing not only data literals, URI references and blank node identifiers as *subject* and *object* arguments (called `subjectExpr` and `valueExpr` in Figure 2), but also variables.

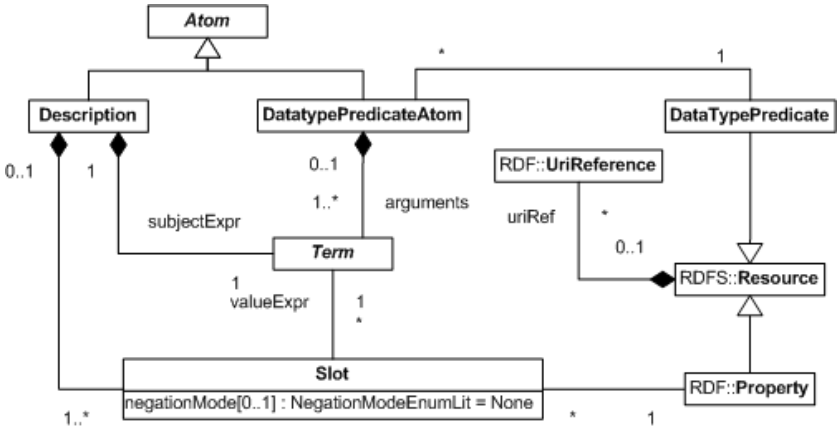


Fig. 2. ERDF Descriptions

An ERDF description consists of the following components:

– One *subject expression*, denoted by the `subjectExpr` property in the metamodel diagram, being an *ERDF term*, that is a `URIReference`, a `Variable`,

an `ExistentialVariable` (blank node identifier) or `rdfs:Literal` (see Figure 3 for the definition of ERDF term).
– A non-empty set of *slots* being property-value pairs consisting of a URI reference denoting a property and an ERDF term as the value expression.

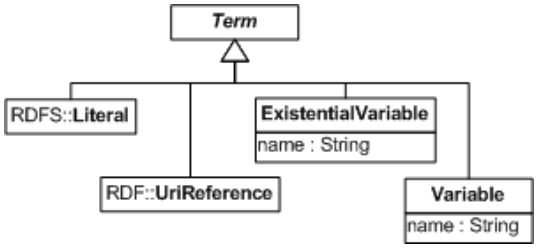


Fig. 3. ERDF Terms

Obviously, descriptions with just one slot correspond to the usual concept of an atomic statement (or triple), while descriptions with multiple slots correspond to conjunctions of such statements. However, as can be seen in Figure 2, all descriptions are considered as ***ERDF atoms***, which in addition subsume *datatype predicate* atoms (datatype predicates are often also called 'built-ins').

ERDF fact statements are variable-free ERDF descriptions such that no slot has a negation mode other than `None` or `Sneg`. That is, only strong negation may occur in fact statements (in the case of negative information).

ERDF descriptions with variables correspond to conjunctive query formulas that can be used as rule conditions.

2.3 ERDF Rules

The abstract syntax of ERDF rules is defined in the metamodel diagram in Figure 4. ERDF rules are derivation rules of the form $D \leftarrow A_1, \ldots, A_n.$ where D is an ERDF description with only `None` or `Sneg` as slot negation modes and $A_1, \ldots, A_n$ are *ERDF atoms*, that is, descriptions or datatype predicate atoms.

3 A Concrete Syntax for ERDF

Our approach is to follow the RDF/XML syntax as much as possible and derive an RDF-style syntax for ERDF atomic formulas, "triple patterns" from the abstract syntax metamodel presented above.

3.1 Expressing a Vocabulary in ERDF

Using the ERDF predicate categories defined in section 2.1, we can refine the FOAF vocabulary definition of `foaf:member`, `foaf:knows` and `foaf:topic_interest` as follows:

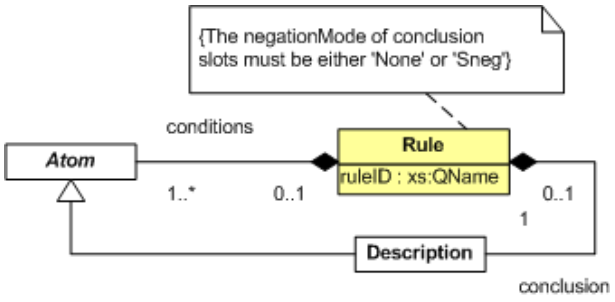


Fig. 4. ERDF Rule

```
<erdf:OpenProperty rdf:about="http://xmlns.com/foaf/0.1/knows">
 <rdfs:domain rdf:resource="http://xmlns.com/foaf/0.1/Person"/>
 <rdfs:range rdf:resource="http://xmlns.com/foaf/0.1/Person"/>
</erdf:OpenProperty>

<erdf:PartialProperty rdf:about="http://xmlns.com/foaf/0.1/topic_interest">
 <rdfs:domain rdf:resource="http://xmlns.com/foaf/0.1/Person"/>
 <rdfs:range rdf:resource="http://www.w3.org/2002/07/owl#Thing"/>
</erdf:PartialProperty>

<erdf:ClosedProperty rdf:about="http://xmlns.com/foaf/0.1/member">
 <rdfs:domain rdf:resource="http://xmlns.com/foaf/0.1/Group"/>
 <rdfs:range rdf:resource="http://xmlns.com/foaf/0.1/Agent"/>
</erdf:ClosedProperty>
```

We identify `erdf:OpenProperty` with `rdf:Property` and `erdf:OpenClass` with `rdfs:Class`. Thus, by default, all RDF predicates are considered to be open. One may argue that is no need of `erdf:OpenProperty` and `erdf:OpenClass` constructs, but for names uniformity and expressivity these constructs are defined as part of the ERDF vocabulary.

3.2 Expressing ERDF Terms

ERDF terms are URI references, blank node identifiers, variables or data literals. They are expressed in two ways, depending on their occurrence as *subject expressions* or as *value expressions*.

Terms as subject expressions are values of the `erdf:about` attribute, which may be URI references, blank node identifiers or variables (using the SPARQL syntax for blank node identifiers and variables).

Terms as value expressions are expressed either with the help of one of the attributes `rdf:resource`, `rdf:nodeID` or `erdf:variable`, or as the text content of the property-value slot element in the case of a data literal.

3.3 Descriptions and Datatype Predicate Atoms

ERDF descriptions are encoded by means of the `erdf:Description` element. Each description contains a non-empty list of (possibly negated) property-value slots.

Example 1. Gerd knows Adrian, has some topic interest, but is not interested in the topic 'motor sports'

```
<erdf:Description erdf:about="#Gerd">
  <foaf:knows rdf:resource="#Adrian"/>
  <foaf:topic_interest rdf:nodeID="x"/>
  <foaf:topic_interest erdf:negationMode="Sneg"
                       rdf:resource="urn:topics:motor_sports"/>
</erdf:Description>
```

`erdf:Description`, as an extension of `rdf:Description` element, allows negated slots and two other possible values for triples subject: variables and literals (as values of `erdf:about` attribute). For expressing RDF triples it is possible to use any of `rdf:Description` or `erdf:Description` elements.

Datatype predicate atoms are n-ary logical atoms. The value of `erdf:arguments` property represent an ordered list of arguments. The `erdf:predicate` XML attribute encodes the URI reference to the predicate.

Example 2. Using built-ins

```
<erdf:DatatypePredicateAtom erdf:predicate="swrlb:add">
 <erdf:arguments>
  <erdf:Variable>?sum</erdf:Variable>
  <rdfs:Literal rdf:datatype="xs:int">40</rdfs:Literal>
  <rdfs:Literal rdf:datatype="xs:int">20</rdfs:Literal>
 </erdf:arguments>
</erdf:DatatypePredicateAtom>
```

3.4 Rules and Rulesets

Two syntaxes for ERDF rules are proposed: (1) a more concise non-XML syntax based on SPARQL triple patterns, and (2) an XML-based syntax, which is useful for rule transformations and interchange.

To express ERDF rules in XML, constructs from R2ML[14] rule markup language are used. Later, it may be an option to use the W3C rule interchange format.

Inspired by Jena Rules[4], the non-XML syntax for ERDF rules is based on SPARQL triple patterns: universal quantified variables prefixed by the '?' symbol, literals, typed literals, URI's or QNames to denote full URI's. Five types of atoms can be used:

[4] Jena Rules Syntax - `http://jena.sourceforge.net/inference/#rules`

– *built-ins*, available in a predefined set [5] and offering the possibility of defining new ones (e.g. `sum(?a,?b,?c)` bound `c` to the value of sum from `a` and `b`).
– *positive triples*, formally expressed as `(subject predicate object)`, e.g. `(ex:John foaf:knows ex:Tom)`;
– *strong negated triples*, denoted by adding the '-' symbol in front of the second node, namely `predicate`, e.g. `(?x -foaf:topic_interest ?t)`;
– *weak-negated triples*, expressed as a built-in, namely `naf`. It's arguments, are the triple's nodes, i.e. : `naf(?x foaf:knows ex:Tom)`.
– *negation-as-failure over strong negation*, the '-' symbol is added in front of the second argument, namely the `predicate`, when the `naf` built-in is used, e.g. `naf(?x -foaf:topic_interest ?t)`.

4 The ERDF Application Programming Interface

The ERDF Application Programming Interface was implemented as an extension of Jena Rules. An extended rule syntax was defined for allowing the two ERDF negation connectives. The rule language is backward compatible, therefore the Jena rules are also supported.

Adding support for reasoning in top of ERDF facts has required modifications in the structure of the Jena API. In Figure 5 are reflected some important changes of the informational model. These improvements were made for allowing representation of negated triples and for dealing with these triple types. The ERDF reasoner was defined as an extension of the Jena backward engine.

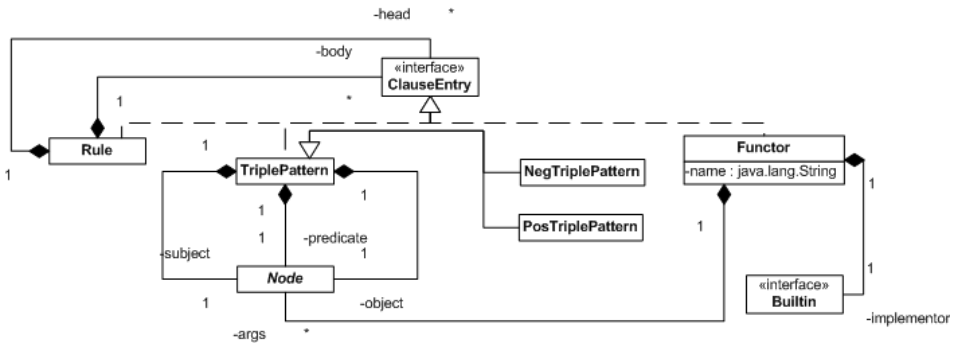


Fig. 5. ERDF Triples extension

The RDF(S)-based reasoner implemented by the Jena API uses an internal set of axioms and rules. For instance, the following axioms are used to express relations used by the RDF Schema:

```
-> (rdf:type      rdfs:range rdfs:Class).
-> (rdfs:Resource  rdf:type  rdfs:Class).
```

[5] Jena built-ins - `http://jena.sourceforge.net/inference/#RULEbuiltins`

An internal set of rules is used for computing the transitive closure in RDF(S). As an example, the following rule consider the `subClassOf` relationship:

```
[(?a rdfs:subClassOf ?b), (?b rdfs:subClassOf ?c) -> (?a rdfs:subClassOf ?c)]
```

ERDF defines `erdf:Class` as being the superclass of all its classes and `erdf:Property` the superclass of all properties. This was reflected by extending the Jena axioms set:

```
-> (rdf:type rdfs:range erdf:Class).
-> (rdfs:Resource rdf:type erdf:Class).
-> (erdf:TotalClass rdfs:subClassOf erdf:Class)
-> (erdf:PartialClass rdfs:subClassOf erdf:Class)
-> (erdf:ClosedClass rdfs:subClassOf erdf:TotalClass)
-> (erdf:OpenClass rdfs:subClassOf erdf:TotalClass)
-> (erdf:TotalProperty rdfs:subClassOf erdf:Property)
-> (erdf:PartialProperty rdfs:subClassOf erdf:Property)
-> (erdf:ClosedProperty rdfs:subClassOf erdf:TotalProperty)
-> (erdf:OpenProperty rdfs:subClassOf erdf:TotalProperty)
-> (erdf:OpenClass rdfs:subClassOf rdfs:Class)
-> (rdfs:Class rdfs:subClassOf erdf:OpenClass)
-> (erdf:OpenProperty rdfs:subClassOf rdf:Property)
-> (rdf:Property rdfs:subClassOf erdf:OpenProperty)
```

Since ERDF deals also with closed properties, the internal rules set was extended to support this feature:

```
[close1: (?s -?p ?o)
         <-
         (?p rdf:type erdf:ClosedProperty)
         (?p rdf:range ?r)(?p rdf:domain ?d)
         (?s rdf:type ?d)(?o rdf:type ?r) naf(?s ?p ?o)]
```

Some other information about the ERDF API might be accessed on the ERDF Web Page[6]. An AJAX based Web Application [7] is provided for testing ERDF rules. The application needs as input data: (1) a set of RDF/ERDF facts (using XML/RDF syntax), (2) a set of rules (using Jena Rules extended syntax), and (3) a set of queries (expressed by using Jena Rules extended syntax). The input data is processed by the ERDF API and query results are returned.

5 Case Study - Building FOAF-Based Working Groups

This section presents a scenario involving FOAF data and ERDF rules. As a short story, an organizing committee needs to create working groups with peoples from different communities taking part at some meeting. The assumption is that every member has his own FOAF file where their topic interests and contacts are provided. The FOAF files are available to the organizing committee.

[6] ERDF - `http://oxygen.informatik.tu-cottbus.de/rewerse-i1/?q=ERDF`
[7] ERDF Rules frontend - `http://oxygen.informatik.tu-cottbus.de/JenaRulesWeb`

The organizers task is to offer a solution (or more) for grouping participating members, having different topic interests areas, by their common interests. Therefore, two members are considered qualified for the same group if they have at least one common interest, but no contradictory interests. The second goal is to extend the community by grouping those members which does not know yet one each other. The meaning of "contradictory topic interest" between two members is that one topic interest of a member is negated in the FOAF file of the other member. The `foaf:knows` property express possible contacts between participants, and `foaf:topic_interest` property denotes interest topics of the meeting participants.

For instance, the organizers have collected the following data from FOAF files of some meeting participants:

```
<rdf:RDF xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
         xmlns:foaf="http://xmlns.com/foaf/0.1/"
         xmlns:erdf="http://www.informatik.tu-cottbus.de/IT/erdf#">

 <erdf:Description erdf:about="http://www.tu-cottbus.de/staff#Gerd">
   <rdf:type rdf:resource="http://xmlns.com/foaf/0.1/Person"/>
   <foaf:topic_interest rdf:resource="urn:topics:RDF"/>
   <foaf:topic_interest rdf:resource="urn:topics:AgentBasedSimulation"/>
   <foaf:topic_interest erdf:negationMode="Sneg"
                        rdf:resource="urn:topics:motor_sports"/>
 </erdf:Description>

 <rdf:Description rdf:about="http://www.ics.forth.gr/staff#Grigoris">
   <rdf:type rdf:resource="http://xmlns.com/foaf/0.1/Person"/>
   <foaf:knows rdf:resource="http://www.tu-cottbus.de/staff#Gerd"/>
   <foaf:topic_interest rdf:resource="urn:topics:RDF"/>
 </rdf:Description>

 <rdf:Description rdf:about="http://www.tu-cottbus.de/staff#Adrian">
   <rdf:type rdf:resource="http://xmlns.com/foaf/0.1/Person"/>
   <foaf:topic_interest rdf:resource="urn:topics:RDF"/>
   <foaf:topic_interest rdf:resource="urn:topics:motor_sports"/>
 </rdf:Description>

 <rdf:Description rdf:about="http://www.tu-cottbus.de/staff#Mircea">
   <rdf:type rdf:resource="http://xmlns.com/foaf/0.1/Person"/>
   <foaf:knows rdf:resource="http://www.tu-cottbus.de/staff#Adrian"/>
   <foaf:topic_interest rdf:resource="urn:topics:AgentBasedSimulation"/>
 </rdf:Description>
</rdf:RDF>
```

Notice that only the first description, since it includes negative triples, needs to be marked up as an ERDF description. For the other (positive) fact statements there is possible to use RDF (but also ERDF).

The following rule, expressed by using XML-based syntax, is defined to establishes if two persons (meeting participants) classify for the same group.

If persons X and Y do not know each other, they have at least one common topic interest and have no contradictory topic interest, then it is recommended that X and Y are members of the same group.

```
<r2ml:DerivationRule r2ml:ruleID="sameGroupAs">
 <r2ml:conditions>
  <erdf:Description erdf:about="?x">
   <rdf:type rdf:resource="foaf:Person"/>
   <foaf:topic_interest erdf:variable="?t"/>
   <foaf:knows erdf:negationMode="Naf" erdf:variable="?y"/>
   <conf:contradictoryInterest erdf:negationMode="Naf" erdf:variable="?y"/>
  </erdf:Description>
  <erdf:Description erdf:about="?y">
   <rdf:type rdf:resource="foaf:Person"/>
   <foaf:topic_interest erdf:variable="?t"/>
   <foaf:knows erdf:negationMode="Naf" erdf:variable="?x"/>
   <conf:contradictoryInterest erdf:negationMode="Naf" erdf:variable="?x"/>
  </erdf:Description>
 </r2ml:conditions>
 <r2ml:conclusion>
  <erdf:Description erdf:about="?x">
   <conf:sameGroupAs erdf:variable="?y"/>
  </erdf:Description>
 </r2ml:conclusion>
</r2ml:DerivationRule>
```

The non-XML syntax for ERDF rules can be used to express the same rule:

```
[sameGroup: (?x conf:sameGroupAs ?y)
            <-
            (?x rdf:type foaf:Person)(?y rdf:type foaf:Person)
            naf(?x foaf:knows ?y) naf(?y foaf:knows ?x)
            naf(?x conf:contradictoryInterest ?y)
            naf(?y conf:contradictoryInterest ?x)]
```

The following rule define how the values of the `conf:contradictoryInterest` predicate are computed:

```
[conInterest: (?p1 conf:contradictoryInterest ?p2)
              <-
              (?p1 rdf:type foaf:Person)(?p2 rdf:type foaf:Person)
              (?p1 foaf:topic_interest ?t)(?p2 -foaf:topic_interest ?t)]
```

Other rules/queries are then used to create `foaf:Groups`. Values computed for `conf:sameGroupAs` property might by considered for this purpose. Consider the following queries:

```
@prefix btu: http://www.tu-cottbus.de/staff#
[q1: <- (btu:Gerd conf:sameGroupAs btu:Mircea)]
[q2: <- (btu:Gerd conf:sameGroupAs btu:Adrian)]
[q3: <- (?p1 conf:sameGroupAs ?p2)]
```

The answer is "true" for *q1* (no contradictory interests and persons does not know each other) and "false" for *q2* ("motor sports" is a contradictory interest). The last query (*q3*) will return all possible combinations of two persons which might be in the same group.

This use case is available for online testing by using the AJAX frontend. The Figure 6 shows an results excerpt obtained by using the above facts, rules and queries as input data in the frontend.

Fig. 6. Query result using ERDF API

6 Related Work

Variables in triples have also been introduced in languages such as N3 [4] and *Jena Rules* [11]. A form of negation-as-failure has been implemented in Jena Rules by using a special built-in predicate. In N3, there is also a form of negation-as-failure, which allows one to test for what a formula does not say, with the help of `log:notIncludes`. But neither N3 nor Jena Rules has a systematic treatment of negative information and open and closed predicates.

7 Conclusion and Future work

The paper presents an abstract and an RDF-style concrete syntax for ERDF, allowing to represent negative fact statements and supports reasoning with open and closed predicates. We have argued that these issues are of practical significance by showing how they affect the popular FOAF vocabulary. Finally a prototype of the ERDF API is described and a practical use case is considered.

Future work includes further extensions of the language, constructs for handling uncertainty and reliability, and their implementation in the ERDF API.

References

1. Anastasia Analyti, Grigoris Antoniou, Carlos Viegas Damasio, and Gerd Wagner. Negation and Negative Information in the W3C Resource Description Framework. *Annals of Mathematics, Computing and Teleinformatics*, 1(2):25–34, 2004.
2. Anastasia Analyti, Grigoris Antoniou, Carlos Viegas Damasio, and Gerd Wagner. Stable Model Theory for Extended RDF Ontologies. In Yolanda Gil, Enrico Motta, V. Richard Benjamins, and Mark A. Musen, editors, *Proceedings of the 4th International Semantic Web Conference*, volume 3729 of *Lecture Notes in Computer Science (LNCS)*, pages 21–36, Galway, Ireland, 6-10 November 2005. Springer-Verlag.
3. Grigoris Antoniou and Frank Van Harmelen. *A Semantic Web Primer*. MIT Press, 2004.
4. Tim Berners-Lee. N3 (Notation 3). http://www.w3.org/DesignIssues/Notation3.html, 1998.
5. D. Brickley and R.V. Guha. RDF Vocabulary Description Language 1.0: RDF Schema. W3C Recommendation February 2004. http://www.w3.org/TR/rdf-schema/.
6. Dan Brickley and Libby Miller. FOAF Vocabulary Specification 0.91. http://xmlns.com/foaf/spec/, November 2007.
7. Klyne G. and Caroll J.J. Resource Description Framework (RDF): Concepts and Abstract Syntax. W3C Recommendation 10 February 2004. http://www.w3.org/TR/rdf-concepts/.
8. Object Management Group. Ontology Definition Metamodel. http://www.omg.org/docs/ptc/07-09-09.pdf, November 2007.
9. Heinrich Herre, Jan O. M. Jaspars, and Gerd Wagner. Partial Logics with Two Kinds of Negation as a Foundation for Knowledge-Based Reasoning. In D.M. Gabbay and H. Wansing, editors, *What is Negation?* Kluwer Academic Publishers, 1999.
10. Peter F. Patel-Schneider, Patrick Hayes, and Ian Horrocks. OWL Web Ontology Language. Semantics and Abstract Syntax. http://www.w3.org/TR/owl-semantics/, February 2004.
11. Dave Reynolds. Jena Rules experiences and implications for rule use cases. In *W3C Workshop on Rule Languages for Interoperability*, 2005.
12. Gerd Wagner. A database needs two kinds of negation. In B. Talheim and H.D. Gerhardt, editors, *3rd Symposium on Mathematical Fundamentals of Database and KnowledgeBase Systems*, volume 495 of *Lecture Notes in Computer Science (LNCS)*, pages 357–371. Springer-Verlag, 1991.
13. Gerd Wagner. Web rules need two kinds of negation. In F. Bry, N. Henze, and J. Maluszynski, editors, *Principles and Practice of Semantic Web Reasoning, Proceedings of the 1st International Workshop, PPSWR '03*, volume 2901 of *Lecture Notes in Computer Science (LNCS)*, pages 33–50. Springer-Verlag, 2003.
14. Gerd Wagner, Adrian Giurca, and Sergey Lukichev. A General Markup Framework for Integrity and Derivation Rules. In F. Bry, F. Fages, M. Marchiori, and H. Ohlbach, editors, *Dagstuhl Seminar Proceedings 05371*, Principles and Practices of Semantic Web Reasoning, 2005.

A Preliminary Report on Answering Complex Queries related to Drug Discovery using Answer Set Programming

Olivier Bodenreider[1], Zeynep H. Çoban[2], Mahir C. Doğanay[3]
Esra Erdem[4], and Hilal Koşucu[5]

[1] National Library of Medicine, National Institutes of Health, USA
[2] Department of Biostatistics, Harvard School of Public Health, USA
[3] Dept. of Mathematics and Computing Science, University of Groningen, The Netherlands
[4] Faculty of Engineering and Natural Sciences, Sabancı University, Turkey
[5] Department of Computer Science, University of Toronto, Canada

Abstract. We introduce a new method for integrating relevant parts of knowledge extracted from biomedical ontologies and answering complex queries related to drug safety and discovery, using Semantic Web technologies and answer set programming. The applicability of this method is illustrated in detail on some parts of existing biomedical ontologies. Its effectiveness is demonstrated by computing an answer to a real-world biomedical query that requires the integration of NCBI Entrez Gene and the Gene Ontology.

1 Introduction

Improvements in Web technologies have brought about various forms of data, and thus WWW has been a huge and easy-to-reach source of knowledge. Particularly recent advances in health and life sciences (e.g., human genome project) have led to generation of a large amount of data. In order to facilitate access to its desired parts, such a big mass of data has been stored in structured forms (like databases or ontologies). For instance, some data/information about drugs is being stored in ontologies, like DRUGBANK and PHARMGKB, available on WWW; and the genes targeted by the drug Epinephrine can be found by searching such a drug ontology using the keyword "Epinephrine."

On the other hand, storing heterogeneous data independent from each other and at different locations has made it difficult to automate high-level reasoning about the stored data. For instance, it is possible to find an answer to the query "What are the genes targeted both by Epinephrine and by Isoproterenol?" only after several steps: considering that a drug (and also a gene) might have been stored in different ontologies under different names, first for each drug a list of genes targeted by that drug could be found, and next these two lists of genes are compared to identify the common ones, by comparing these two lists of genes. Such complex queries, which require appropriate integration of knowledge stored in different places and in various forms, can be answered by current Web technologies most of the time only by some direction/reasoning of humans. This slows down vital research, like drug discovery, that requires comparative data analysis and high-level reasoning and decision making.

Motivated by these challenges, this paper studies the problem of integrating various data sources to be able to perform high-level reasoning tasks, including answering

complex queries using both Semantic Web technologies and Answer Set Programming (ASP) [1–4]. The idea is to build a rule layer using ASP over ontologies described with some Semantic Web technologies. The rule layer not only provides rules to link parts of the ontologies but also provides some background knowledge to be able to perform various reasoning tasks, such as query answering.

That most of the information about biomedical ontologies are actually defaults and that most biomedical ontologies contain incomplete knowledge motivated us to use a nonmonotonic formalism to build a rule layer over ontologies. That experts might want to express preferences as well as constraints while querying the knowledge stored in ontologies to be able to discover new knowledge, and that ASP provides an expressive language to express them and efficient solvers, like DLVHEX[6] [5] built over DLV,[7] to reason about them motivated us to use ASP as such a nonmonotonic formalism.

2 Three Ontologies

To experiment with our ASP approach to integrating biomedical ontologies and reasoning about them, and to illustrate its applicability, we have developed three ontologies, namely a gene ontology, a disease ontology, and a drug ontology. We have built these ontologies from existing knowledge from various data sources available on the Web. These ontologies are written in RDF(S). To develop our disease ontology, first we selected a set of diseases. The names (and their synonyms) of each disease are taken from PHARMGKB database.[8] Information about the symptoms of these diseases is obtained from the Medical Symptoms and Signs of Disease web page.[9] Information about the genes related to each disease are also extracted from PHARMGKB. Each disease is classified in some category relative to the information available at the Genes and Diseases web page.[10] Some components of the disease ontology is shown in Table 1. We have prepared the other two ontologies in a similar way, using PHARMGKB, UNIPROT,[11] GENE ONTOLOGY (GO),[12] GENENETWORK database,[13] DRUGBANK,[14] and the Medical Symptoms and Signs of Disease web page.

3 Integrating Knowledge Extracted from Different Ontologies

DLVHEX provides constructs to import external theories that may be in different formats. For instance, consider as an external theory our drug ontology described in RDF. All triples from this theory can be exported using the external predicate `&rdf`:

[6] `http://con.fusion.at/dlvhex/`
[7] `http://www.dbai.tuwien.ac.at/proj/dlv/`
[8] `http://www.pharmgkb.org/.`
[9] `http://www.medicinenet.com/symptoms_and_signs/article.htm.`
[10] `http://www.ncbi.nlm.nih.gov/disease/.`
[11] `http://www.ebi.uniprot.org/index.shtml.`
[12] `http://www.geneontology.org.`
[13] `http://humgen.med.uu.nl/~lude/genenetwork/.`
[14] `http://redpoll.pharmacy.ualberta.ca/drugbank/.`

Table 1. The disease "Asthma" described in our disease ontology

has_name	Asthma
has_synonyms	Bronchia, Bronchial Asthma
has_symptoms	Coughing, Wheezing, Chest tightness, Shortness of breath, Faster breathing
related_genes	ABCC1, ADA, ADAM33, ADCY9, AD0RA1, ADRB1, ADRB2, ALOX5, COMT, CRHR1
treatedBy_drugs	Isoproterenol, Flunisolide, Salbutamol

```
triple_drug(X,Y,Z) :- &rdf["URI for Drug Ontology"](X,Y,Z).
```

Not all triples may be relevant to the query asked by the user. For instance, if one asks for the names of drugs listed in the ontology, then only the triples that describe the names of drugs are sufficient to answer this query. The names of drugs, out of all properties about drugs described in `drug.rdf`, can be extracted by the following rule:

```
drug_name(A) :- triple_drug(_,"drugproperties:name",A).
```

If the query were about gene-gene interactions, then we could extract the relevant part of the gene ontology by the rules

```
gene_gene(G1,G2) :- triple_gene(X,"geneproperties:name",G1),
   triple_gene(X,"geneproperties:related_genes",B),
   triple_gene(B,Z,Y), Z!="rdf:type",
   triple_gene(Y,"geneproperties:name",G2).
```

Once necessary parts of ontologies are extracted from ontologies, one can define further concepts to integrate these knowledge. For instance, once we extract the gene-gene interactions, we can obtain all chains of gene-gene interactions for a gene targeted by a drug, by defining the transitive closure of `gene_gene`:

```
tc_gene_gene(X,Y) :- gene_gene(X,Y).
tc_gene_gene(X,Y) :- gene_gene(X,Z), tc_gene_gene(Z,Y).
```

Now let us relate this information to a gene G targeted by a drug D by finding every gene $G1$ that is related to G by means of a chain of interactions:

```
drugTargetedGene_interacts_gene(D,G,G1) :-
   drug_targets(D,G), tc_gene_gene(G,G1).
```

4 Answering Complex Queries using DLVHEX

With the help of Devrim Gözüaçık (a medical doctor and a molecular biologist), we have identified a set of meaningful queries about drugs, genes, diseases, towards drug safety and discovery. We present here only three of them:

Q6 What are the sideeffects that are shared by all the drugs that treat a disease D?

Q12 Is there a drug that has no toxicity information?
Q14 Does a drug R alleviate at least 1 symptom of a disease D and have at most 2 symptoms of D as side effects?

We integrate relevant parts of ontologies, and formulate these queries as follows.

Q6 What are the sideeffects that are shared by all the drugs that treat a disease D?

For the disease `Asthma`, this query can be formulated as follows:

```
answer :- sideeffect(S), common_sideeffect("Asthma",S).
:- not answer.
```

Here `common_sideeffect` is defined as follows:

```
-common_sideeffect(D,S) :- not drug_sideeffect(R,S),
   drug_disease(R,D), sideeffect(S).

common_sideeffect(D,S) :- not -common_sideeffect(D,S),
   sideeffect(S), disease_name(D).
```

Here is a part of the answer DLVHEX finds to the query above:

```
flushing dizziness headache
```

Q12 Is there a drug that has no toxicity information?

To answer this query, we define a new concept of "unknown" toxicity:

```
unknown_toxicity_drug(X) :- drug_synonym(R,X),
   not drug_istoxic(R), not -drug_istoxic(R).
```

where `drug_istoxic(R)` describes that the drug `R` is toxic, and `-drug_istoxic(R)` describes that the drug `R` is not toxic:

```
drug_istoxic(R) :- triple_drug(X,"drugproperties:name",R),
   triple_drug(X,"drugproperties:is_toxic","yes").
drug_istoxic(R) :- drug_synonym(R,R1), drug_istoxic(R1).

-drug_istoxic(R) :- triple_drug(X,"drugproperties:name",R),
   triple_drug(X,"drugproperties:is_toxic","no").
-drug_istoxic(R) :- drug_synonym(R,R1), -drug_istoxic(R1).
```

For the query

```
:- not unknown_toxicity_drug("Isoproterenol").
```

DLVHEX returns an answer set; therefore the answer to the query above is positive.

Q14 Does a drug R alleviate at least 1 symptom of a disease D and have at most 2 symptoms of D as side effects?

To answer this query we define a new concept:

```
a_drug_disease_relation(R,D) :-
   disease_name(D), drug_name(R),
   1 <= #count{S:drug_symptom(R,S),disease_symptom(D,S)},
   #count{S:drug_sideeffect(R,S),disease_symptom(D,S)}<=2.
```

For the query

```
:- not a_drug_disease_relation("Isoproterenol",
                               "Substance Related Disorders").
```

DLVHEX returns no answer set; therefore the answer to the query above is negative.

5 From *Glycosyltransferase* to *Congenital Muscular Dystrophy*

To investigate the effectiveness of our approach to answering real-world queries, we have considered a slight modification of the complex query studied in [6]:

> Find all the genes annotated with the molecular function *glycosyltransferase* or any of its descendants and associated with any form of *congenital muscular dystrophy*.

and tried to reproduce the same results. In the query of [6] the GO ID for glycosyltransferase is given. The query above requires integration of NCBI Entrez Gene (EG) and the Gene Ontology (GO).

To find an answer to this query, we have used the RDF version of GO that is released on February 6, 2008; it contains 416700 RDF triples. We have used an RDF version of EG that contains 673180 RDF triples.

The computation of an answer consists of two parts: extracting relevant knowledge from each ontology and integrating them. We have extracted from GO the molecular function *glycosyltransferase* and its descendants by the rules

```
mf_isa(Y) :- triple_go(Y,"go:name",YN),
   &strstr[YN,"glycosyltransferase"].
mf_isa(Y) :- triple_go(Y,"go:synonym",YN),
   &strstr[YN,"glycosyltransferase"].

mf_isa(X) :- triple_go(X,"go:is_a",Y), mf_isa(Y).
mf_isa(X) :- triple_go(X,"go:synonym",XN),
   triple_go(Z,"go:name",XN), triple_go(Z,"go:is_a",Y), mf_isa(Y).
```

The first two rules extract the molecular functions whose names or synonyms contain the string "glycosyltransferase". The last two rules extract the descendants of these molecular functions, considering their synonyms.

Similarly, we have extracted from EG the diseases with any form of *congenital muscular dystrophy*, by the rules

```
gene_disease(Y,D) :- triple_eg(Y,"eg:has_OMIM_record",Z),
   triple_eg(Z,"eg:has_textual_description",D),
   &strstr[D,"congenital"], &strstr[D,"muscular"],
   &strstr[D,"dystrophy"].
```

After that we have integrated the extracted knowledge by the rules

```
gene_mf_disease(Y,XI,D) :- gene_disease(Y,D),
   triple_eg(Y,"eg:has_GeneOntology_annotation",X),
   mf_isa(XI), triple_eg(X,"eg:has_GO_ID",XI).
```

and computed the following answer (the same as in [6]) to the query:

```
gene_mf_disease("http://www.ncbi.nlm.nih.gov/dtd/NCBI_Entrezgene.
  dtd/9215", "http://www.geneontology.org/go#GO:0008375",
  "Muscular dystrophy, congenital, type 1D")
```

DLVHEX extracts relevant knowledge from the ontologies, integrates them, and computes the answer above in 9 minutes, on a machine with Intel Centrino 1.8GHz CPU and 1 GB of RAM running on Windows XP.

6 Conclusion

We have studied integrating relevant parts of knowledge extracted from biomedical ontologies, and answering complex queries related to drug safety and discovery, using Semantic Web technologies and Answer Set Programming (ASP). We have illustrated the applicability of this method on some ontologies extracted from existing biomedical ontologies, and its effectiveness by computing an answer to a real-world biomedical query that requires the integration of NCBI Entrez Gene and the Gene Ontology. We have also compared our approach with the existing Semantic Web technologies that support representing and answering queries. We have observed about these technologies that, due to lack of support for rules or for some concepts (e.g., transitive closure, negation as failure, cardinality constraints), some queries can not be represented concisely and some queries can not be represented at all. In this sense, the ASP-approach provides a more expressive formalism to represent rules, concepts, constraints, and queries.

Acknowledgments

Devrim Gözüaçık helped us identify some of the complex queries. Thomas Krennwallner and Roman Schindlauer helped us with installing/using DLVHEX. RACER Systems provided us a free, educational version of RACERPRO,[15] to be used in connection with DLVHEX. Anonymous reviewers provided useful comments on an earlier draft. This research was supported in part by the Intramural Research Program of the National Institutes of Health (NIH), National Library of Medicine (NLM).

References

1. Lifschitz, V.: Action languages, answer sets and planning. In: The Logic Programming Paradigm: a 25-Year Perspective. Springer (1999)
2. Marek, V., Truszczyński, M.: Stable models and an alternative logic programming paradigm. In: The Logic Programming Paradigm: a 25-Year Perspective. Springer (1999)
3. Niemelä, I.: Logic programs with stable model semantics as a constraint programming paradigm. Annals of Mathematics and Artificial Intelligence **25** (1999)
4. Baral, C.: Knowledge Representation, Reasoning and Declarative Problem Solving. Cambridge University Press (2003)
5. Eiter, T., G.Ianni, R.Schindlauer, H.Tompits: Effective integration of declarative rules with external evaluations for Semantic-Web reasoning. In: Proc. of ESWC. (2006)
6. Sahoo, S.S., Zeng, K., Bodenreider, O., Sheth, A.: From "glycosyltransferase" to "congenital muscular dystrophy": Integrating knowledge from NCBI Entrez Gene and the Gene Ontology. In: Proc. of Medinfo. (2007)

[15] `http://www.racer-systems.com/`.