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].

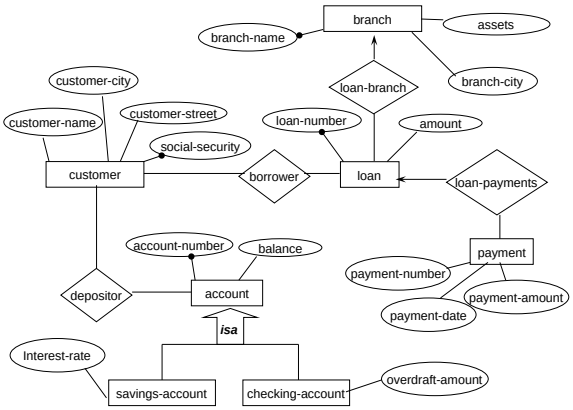

Fig. 1. The Banking Enterprise ER diagram

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:

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: $f("Spagna") : branch(name:"Spagna", city:"Rome", assets:1000000)$. In this way, the *functional object identifier* $f("Spagna")$ 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:

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

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, relative 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 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 sufficient 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 axioms 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 software

[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 settings OntoDLP axioms can model both denial constraints (like functional dependencies) and inclusion dependencies (in the latter case, negation as failure is exploited).

developers through 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 *branchPredicate*. 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: $a^*(t) :- a(t), not\ \overline{a}(t)$.
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: $a^r(t) :- a^*(t), not\ \overline{a}(t), not\ \overline{\overline{a}}(t)$.

[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 linking 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 from 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 enterprise 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. Terracina, G., Leone, N., Lio, V., 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/.