Lifting Databases to Ontologies[⋆]

Gisella Bennardo, Giovanni Grasso, Salvatore Maria Ielpa, Nicola Leone, Francesco Ricca

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

Abstract. Nowadays it is widely recognized that ontologies are a fundamental tool for knowledge representation and reasoning; and, in particular, they have been recently exploited for setting out business enterprise information (obtaining the so-called *enterprise/corporate ontologies*). Enterprise ontologies offer a clean view of 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, developing an enterprise ontology by scratch would be a time-consuming and expensive task; and, (*ii*) usually, enterprises already store their relevant information in large database systems, and do not want to load the information again in the ontologies; moreover, these databases have to keep their autonomy since many applications work on them. In this paper we propose a solution that combines the advantages of an ontology representation language (i.e., high expressive power and clean representation of data) having powerful reasoning capabilities, with the capability to efficiently exploit a large (and, often already existent) enterprise database. In particular, we allow to “lift” an existing database to an ontology. The database is kept and the existing applications can still work on it, but the user can take profit of the new ontological view of the data, and exploit powerful reasoning mechanisms for consistency checking, knowledge discovery, and other advanced knowledge-based tasks.

Kewwords: Logic Programming, Answer Set Programming, Ontology Languages, Enterprise Ontologies, Databases.

1 Introduction

Nowadays, 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.

[⋆] 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.”

In this field, *ontologies* [1] have been recognized to be a fundamental tool. Indeed, ontologies are well-suited formal tools to provide a clean abstract specification of the entities 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. Indeed, 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;
(*ii*) usually, enterprises already store their relevant information in large database systems.

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 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 because these databases have to keep their autonomy (considering that many applications work on them).

In this paper we propose a solution that combines the advantages of an ontology representation language (i.e., high expressive power and clean representation of data) having powerful reasoning features, with the capability to efficiently exploit a large (and, often already existent) enterprise database. In particular, we allow to "lift" an existing database to an ontology in the spirit of [2]. The database is kept and the existing applications can still work on it, but the user can take profit of the new ontological view of the data, and exploit powerful reasoning mechanisms for consistency checking, knowledge discovery, and other advanced knowledge-based tasks.

This has been done by properly extending the OntoDLV system [3, 4] language by suitable constructs, called *virtual class* and *virtual relation*, that allows one to specify the instances of an ontology concept/relation when they "already exist" autonomously in a relational database.

More in detail, OntoDLV implements a powerful logic-based ontology representation language, called OntoDLP, which is an extension of (disjunctive) Answer Set Programming [5–7] (ASP) with all the main ontology constructs including classes, inheritance, relations, and axioms. OntoDLP is strongly typed, and it combines in a natural way the modeling power of ontologies with a powerful "rule-based" language.[1]

Suppose now that it is given an existing database; then, we can analyze its schema and comfortably recognize both entities and relationships that the database engineer stored on it, and we can represent it by means of an OntoDLP ontology. This gives us a clean and high level specification of the knowledge present in the given database, but the obtained intensional specification is not linked with it. Since, up to now, both ontology specification and logic programs have to be specified directly in the OntoDLP syntax in order to exploited the ontology, then the database should be loaded into the OntoDLV system (which, as previously pointed out, is inconvenient). To overcome this limitation, we purposely extended the OntoDLV language in order to directly specify how the instances of an OntoDLP class have to be obtained from the given database. In particular, we introduced *virtual classes* (and *virtual relations*), that are classes (and relations) whose instances are specified by means of special logic rules. Those rules admit a special kind of logic predicates which represent database tables/views or even SQL queries results. Basically, those rules define a mapping among the data in the database and the corresponding instances of the ontology. Given that, the obtained ontology specification can be exploited as usual, and all the powerful features on OntoDLP (from advanced type-checking to complex reasoning) can be exploited on existing data.

Moreover, we extended the OntoDLV system in order to implement *virtual classes* and *virtual relations*, in such a way that it seamlessly provide to the users both abstract browsing and query facility on the ontology and efficient query processing on existing data sources.

The remainder of the paper is organized as follows. In the next section, we provide a brief overview of the original OntoDLP language. In Section 3 we show, by means of an example, how an existing database can be lift to an ontology by exploiting virtual classes. Section 4 overviews the architecture and the implementation of virtual classes in the OntoDLV system. Eventually, in Section 5 we draw conclusions and compare related works.

2 The OntoDLP language

In this section we briefly describe OntoDLP, an ontology representation and reasoning language which provides the most important ontological constructs, namely classes, attributes, relations, inheritance and axioms, and combines them with the reasoning capabilities of ASP. For a detailed description refer to [3, 4].

Hereafter, we assume the reader to be familiar with ASP syntax and semantics, for further details refer to [5, 9].

Classes. A *class* can be thought of as a collection of individuals that belong together because they share some properties. Classes can be defined in OntoDLP

[1] 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) [8].

by using the keyword **class** followed by its name. Class attributes can be specified by means of pairs *(attribute-name : attribute-type)*, where *attribute-name* is the name of the property and *attribute-type* is the class the attribute belongs to.

For instance, *person*, *food*, and *place* are classes of individuals, that can be defined in OntoDLV as follows:

$\textbf{\textit{class}}\, place(name\!:\!string).$
$\textbf{\textit{class}}\, food(name\!:\!string,\, origin\!:\!place).$
$\textbf{\textit{class}}\, person(name\!:\!string, father\!:\!person,\, mother\!:\!person,\, birthplace\!:\!place).$

Class attributes in OntoDLP model the properties that *must* be present in all class instances; properties that *might* be present or not might be modeled, for instance, by using relations. Moreover, class definitions can be recursive (e.g., in class person both father and mother are of type *person*), and attribute types can exploit the built-in classes *string* and *integer* (respectively representing the class of all alphanumeric strings and the class of non-negative integers).

Objects. Domains contain individuals which are called *objects* or *instances.* Each individual in OntoDLP belongs to a class and is uniquely identified by a constant called *object identifier* (oid). Objects are declared by asserting a special kind of logic facts (asserting that a given instance belongs to a class). For example, with the fact:

$john : person(name\!:\text{“}John\text{”}, father\!:\!jack,\, mother\!:\!ann,\, birthplace\!:\!rome).$

we declare that *john* is an instances of the class *person.* Note that, when we declare an instance, we immediately give an oid to the instance which may be used to fill an attribute of another object. In the example above, the attribute birthplace is filled with the oid *rome* (identifying an instance of *Place*) modeling the fact that *john* is born in Rome; in the same way, *jack* and *ann* are suitable oids respectively filling the attributes *father*, *mother* (both of type person).

Oids are proper of a given base class, i.e., base classes cannot share individuals. However, an individual may belong to different classes when other two modeling tools are employed (that will be described later), namely: inheritance and collection classes.

Inheritance. Concepts in an ontology are usually organized in taxonomies by using the *specialization/generalization* mechanism (which is called *inheritance* in object-oriented languages). For instance, employees are a special category of person having extra attributes, like *salary* and *company*. OntoDLV supports inheritance by means of the special binary relation *isa*. In particular, the above-mentioned employee class can be declared as follows:

$\textbf{\textit{class}}\, employee\;\; \textbf{\textit{isa}}\; \{person\}(salary\!:\!integer,\, boss\!:\!person).$

In this case, *person* is a more generic concept or *superclass* and *employee* is a specialization (or *subclass*) of *person*. Moreover, an instance of *employee* will have the local attributes *salary* and *boss*, in addition to those that are defined in *person*. We say that the latter are “inherited” from the superclass *person*. Hence, each proper instance of *employee* will also be automatically considered an instance of *person* (the opposite does not hold!).

Collection Classes The notions of base class and base relation introduced above correspond, from a database point of view, to the *extensional* part of the OntoDLP language. However, there are many cases in which some property or some class of individuals can be "derived" (or inferred) from the information already stated in an ontology. In particular, OntoDLP allows one to specify the instances of a class by means of logic rules, thus obtaining a *Collection class.*

For instance, the class *richEmployee* can be defined as follows:

$\boldsymbol{collection\ class}\, richEmployee(name\!:\!string)\{$
$E : richEmployee(name\!:\!N)$:– $E : employee(name\!:\!N, salary\!:\!S), S > 1000000.\}$

Basically, this class *collects* instances defined by another class (i.e., *person*) and performs a re-classification based on some information which is already present in the ontology.

Importantly, the programs (set of rules) defining collection classes must be normal and stratified (see e.g., [10, 11]).

Relations. Another important feature of an ontology language is the ability to model relationships among individuals. Relations are declared like classes: the keyword ***relation*** (instead of ***class***) precedes a list of attributes.

As an example, we model a relationship between person and living place as follows:

$\boldsymbol{relation}\, personLivesIn(individual\!:\!person, location\!:\!place).$

The instances of a relation are called *tuples*; for instance, we can assert that *john* lives in *rome* by writing a logic fact as follows:

$personLivesIn(individual\!:\!john, location\!:\!rome).$

Contrary to class instances, tuples are not equipped with an oid.

As for classes, also relations can be defined via rules, obtaining the so-called *intensional relations.*

We complete the description of relations observing that OntoDLP allows one also to organize them in taxonomies. Basically, attributes and tuples are inherited by following the same criterions defined above for classes.

Axioms and Consistency. *Axioms* are a consistency-control construct modeling sentences that are always true. For example, we may enforce that a person cannot be father of itself as follows:

:– $X : person(father\!:\!X).$

If an axiom is violated, we say that the ontology is inconsistent (i.e., it contains information which is contradictory or not compliant with the domain's intended perception).

Reasoning modules and queries. 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 logic 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 queries, on both he ontology and the predicates defined in reasoning modules. Queries offer the possibility of extract knowledge implicitly contained in the ontology. As an example, we ask for the list of person having a father who is born in Rome as follows:

$$X : \mathit{person}(\mathit{father}\!:\!\mathit{person}(\mathit{birthplace}\!:\!\mathit{place}(\mathit{name}\!:\text{``}\mathit{Rome}\text{''})))?$$

3 Virtual Classes and Virtual Relations

In this section we show how an existing database database can be "lift" to an OntoDLP ontology. In particular, the new features of the language, namely, *virtual classes* and *virtual relations*, which have been conceived for dealing with this problem, will be illustrated by exploiting the following example.

Suppose that a Banking Enterprise asks for building an ontology of its domain of interest. This request has arisen from the need of an uniform view of the knowledge stored in the enterprise information system, which is shared among all the enterprise branches. Indeed, ontologies offer a clear high-level perspective of a domain, which can be, thus, analyzed by exploiting more expressive and powerful querying/reasoning methods.

The schema of the existing database exploited by the information system of the banking enterprise is reported in Table 1.

The first step that must be done is to reconstruct the semantics of 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 which describes the structure of the entities stored in the database. Likely, the database engineers exploited the Entity-Relationship Model (ER-model) [12], 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 (there are few well-known rules for obtaining a database schema from an ER-model) 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 an 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 an unique account-number, and maintains last access date. Moreover, accounts can

Table name	**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.

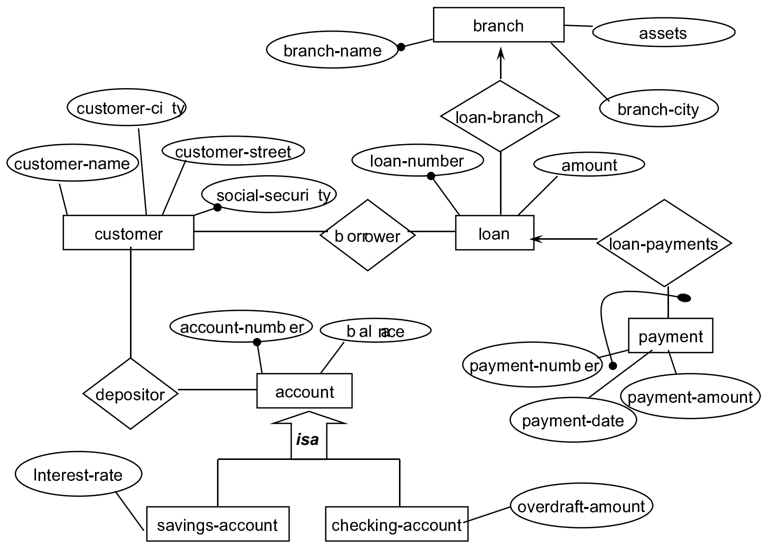

Fig. 1. The Banking Enterprise ER diagram

be held by more than one 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 we obtained so-far, represents a good starting point for defining an ontology that describes the banking enterprise domain. Indeed, we can easily exploit it for identifying ontology concepts and for detecting which tables store the information regarding their instances.

At this point, what we have to do for "lifting" the banking database to a banking ontology is to create an OntoDLP (base) class, with name *c*, for each concept *c* in the domain, and exploit special logic rules to specify a mapping between class *c* and its instances "stored" in the database. A class *c* defined by means of such kind of mapping rules is called *virtual*, because its instances come from an external source but, as far as reasoning and querying are concerned, they appear directly specified in OntoDLP.

Thus, in more detail, a *virtual class* is defined by using the keywords **virtual class** followed by its name, and specifying a list of attributes by means of pairs *(attribute-name : attribute-type)*; while, as previously pointed out, its instances are defined by means of rules, which contain some special predicates that allows one to access the original database tables. For example, we model the *branch* entity as follows:

$$
\begin{array}{l}
\textit{\textbf{virtual class}}\; branch(name\!:\!string,\, city\!:\!string,\, assets\!:\!integer)\{\\
\quad f(BN) : branch(name\!:\!BN,\, city\!:\!BC,\, assets\!:\!A) \;{:}{-}\\
\qquad\quad [db1, \text{"}SELECT\; branch\text{-}name\; AS\; BN,\, branch\text{-}city\; AS\; BC,\, assets\; AS\; A\\
\qquad\qquad FROM\; branch\text{"}]\}
\end{array}
$$

Note how 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 *SQL atom*) which contains an SQL query. More in detail, a *SQL atom* consists of a pair [db object identifier, sql query] enclosed in square brackets. The db object identifier plays a crucial role, because identifies the database on which the sql query will be performed. As a matter of fact, data sources are specified directly in OntoDLP as instances of the built-in class *dbSource* as follows:

$$
\begin{array}{l}
db1 : dbSource(connectionURI\!:\!\text{“}http : //mydb.mysite.com\text{”},\, user\!:\!\text{“}myUser\text{”},\\
\qquad\qquad\qquad\qquad\qquad\qquad password\!:\!\text{“}myPsw\text{”}).
\end{array}
$$

This statement is automatically recognized, since the system automatically provides those declarations:

$$
\begin{array}{l}
\textit{\textbf{class}}\; externalSource\,.\\
\textit{\textbf{class}}\; dbSource\;\; \textit{\textbf{isa}}\; \{externalSource\}(connectionURI\!:\!string,\, user\!:\!string,\\
\qquad\qquad\qquad\qquad\qquad\qquad password\!:\!string).
\end{array}
$$

Note that such a mechanism allows to build an ontology starting from one or more databases, just specifying more *dbSources*. Moreover, this scheme is sufficiently general to be (in the future) extended also to access other kind of sources beside databases.

An important issue to be better described in the above example regards the way how object identifiers for virtual class instances are built. First of all, note that while the database stores values, ontologies manage instances each of which is uniquely identified by an object identifier, which is not merely a value. This is the well-known impedance mismatch problem. We provide a specific mechanisms for facing this problem, in which values appearing in the databases are kept, someway, distinct from object identifiers appearing in the ontology. To this end, all the instances of a *virtual class* are identified by means of a *functional object identifier* that is suitably built from data values stored at the databases. 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 at the table *branch*.[2]

[2] Note that *name* is a key for table *branch*. Since 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; however, in order to obtain the maximum flexibility, the responsibility of writing a “right” ontology mapping is left to the ontology engineer.

In practice, if the *branch* table stores a tuple *(Spagna, Rome, 1000000)*, then the associated instance in the ontology will be:

$f(\mathit{Spagna}) : \mathit{branch}(\mathit{name}\!:\!\mathit{Veneto}, \mathit{city}\!:\!\mathit{Rome}, \mathit{assets}\!:\!\mathit{1000000})$

In this way we build the *functional object identifier* $f(\mathit{Spagna})$ starting from the data value *Spagna*, keeping data values alphabet distinct from the one of *functional object identifiers.*

We say that a *virtual class* declared by means of *SQL atoms* is in *sql notation*, but we provided, in addition, a more direct notation for accessing database tables, called *logical notation.*

In particular, the *virtual class branch* can be equivalently defined as follows:

$\textbf{\textit{virtual class}}\ \mathit{branch}(\mathit{name}\!:\!\mathit{string}, \mathit{city}\!:\!\mathit{string}, \mathit{assets}\!:\!\mathit{integer})\{$
$\quad f(BN) : \mathit{branch}(\mathit{name}\!:\!BN, \mathit{city}\!:\!BC, \mathit{assets}\!:\!A) :-$
$\qquad \mathit{branch}@\mathit{db}1(\mathit{branch\text{-}name}\!:\!BN, \mathit{branch\text{-}city}\!:\!BC, \mathit{assets}\!:\!A).\}$

The *logical notation* differs from the *sql* one by using *sourced atoms* in replacing of *SQL atoms.* A *sourced atoms* consist of a name (*branch*) that identifies a table "at" (@) a specific database source (*db1*), in addition to a list of attribute-names (that must match those in the table) linked to values or variables.

Hereafter, we will use the *logical notation* in all the examples of *virtual classes* and *virtual relations.*

The next entity we focus on is *customer*, for dealing with it we define another virtual class as follows:

$\textbf{\textit{virtual class}}\ \mathit{customer}(\mathit{ssn}\!:\!\mathit{string}, \mathit{name}\!:\!\mathit{string}, \mathit{street}\!:\!\mathit{string}, \mathit{city}\!:\!\mathit{string})\{$
$\quad c(SSN) : \mathit{customer}(\mathit{ssn}\!:\!\mathit{SSN}, \mathit{name}\!:\!N, \mathit{street}\!:\!S, \mathit{city}\!:\!C) :-$
$\qquad \mathit{customer}@\mathit{db}1(\mathit{social\text{-}security}\!:\!\mathit{SSN}, \mathit{customer\text{-}name}\!:\!N, \mathit{customer\text{-}street}\!:\!S,$
$\qquad\qquad \mathit{customer\text{-}city}\!:\!C).\}$

Note that, in this case we used the functional term $c(SSN)$ in order to assign to each instance a suitable *functional object identifier* built on the *social-security* attribute value. Actually, we use one fresh functor for each virtual class; in this way, we are sure that functional object identifiers, belonging to different classes, are distinct. In our example, the *customer* and the *branch* class instances, are thus made disjoint. In fact, the former uses the functor f, while the latter uses the functor c.

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

$\textbf{\textit{virtual class}}\ \mathit{loan}(\mathit{number}\!:\!\mathit{integer}, \mathit{loaner}\!:\!\mathit{branch}, \mathit{amount}\!:\!\mathit{integer})\{$
$\quad l(N) : \mathit{loan}(\mathit{number}\!:\!N, \mathit{loaner}\!:\!f(L), \mathit{amount}\!:\!A) :-$
$\qquad \mathit{loan}@\mathit{db}1(\mathit{loan\text{-}number}\!:\!N, \mathit{branch\text{-}name}\!:\!L, \mathit{amount}\!:\!A).\}$

The above examples is slightly different from the ones so far illustrated. In fact, the *loan* class has an attribute (*loaner*) of type *branch*, which is a virtual class too. In this case we have to carefully deal with functional terms in order to ensure referential integrity. As shown above in our example, we face this conditions by properly using *functional object identifiers.* Note in fact that the mapping uses the functional term $f(L)$ to build values for the *loaner* attribute (rule's head of the *loan* virtual class).

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 this way, we maintain referential integrity constraints unchanged at the ontology level, achieving a consistency-safe results.

In the next example we stress the above idea while modeling the *payment* entity:

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

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 in OntoDLP by exploiting inheritance (see Section 2). Thus, we introduce a *virtual class* named *account* as follows:

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

and, in addition, we provide two *virtual classes savingAccount* and *checkingAccount* both subclasses of *account* which contain the mappings with the corresponding database tables:

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

$$
\begin{array}{l}
\textbf{\textit{virtual class}}\, checkingAccount \;\; \textbf{\textit{isa}} \; \{account\}(overdraft : integer)\{\\
\quad acc(N) : checkingAccount(number : N, balance : B, overdraft : I) \text{ :- }\\
\qquad checking\text{-}account\, @db1(account\text{-}number : N, balance : L, overdraft\text{-}amount : I).\}
\end{array}
$$

Up to now, we specified all the concepts in the banking domain, but we miss model relationship between them. For instance, the ER diagram clearly shows that *customers* and *loans* are in relationship through relations *borrower* and *depositor*. To deal with this problem, OntoDLP allows to define also *virtual relations* besides virtual classes. Hence, we can directly model both *borrower* and *depositor* as follows:

$$
\begin{array}{l}
\textbf{\textit{virtual relation}}\, borrower(cust : customer, loan : loan)\{\\
\quad borrower(cust : c(C), loan : l(L)) \text{ :- }\\
\qquad\qquad borrower@db1(customer\text{-}social\text{-}sec : C, loan\text{-}number : L).\}
\end{array}
$$

$\textbf{\textit{virtual relation}}\ depositor(cust : customer, account : account, , lastAccess : date)\{$
$\quad depositor(cust : c(C), account : acc(A), lastAccess : D) :-$
$\qquad depositor@db1(customer\text{-}social\text{-}sec : C, account\text{-}number : A, access\text{-}date : d).\}$

It is worth noting that a *virtual relation* differs from a *virtual class* mainly because the latter does not specify object identifiers for its instances (tuples). In fact, *virtual relations* represent some properties that link individuals already present in the ontology. However, as previously pointed out, we have to carefully take into account integrity constrains by properly using functional object identifiers.

4 Virtual Entities Implementation

In this Section we describe how *virtual classes* and *virtual relations* have been implemented into the OntoDLV system. To this end, we first describe the general architecture of the system, and then we detail the modules that have been introduced/extended for dealing with the new features. We refrain from giving an in-depth description of all technical details underlying the implementation of OntoDLV, rather we present the main new features of the system.

4.1 OntoDLV Architecture

OntoDLV is a complete framework that allows one to specify, navigate, query and perform reasoning on OntoDLP ontologies.

The system architecture of OntoDLV, depicted in Figure 2, can be divided in three abstraction layers. The lowest layer, named *OntoDLV core* contains the components implementing the main functionalities of the system; above it, the *Application Programming Interface* (API) act as a facade for supporting for the development of applications based on the core; while the Graphical User Interface (GUI) is the end-user interface of the system.

In turn, the OntoDLV core is made of three submodules, namely: *Persistency Manager, Type Checker*, and *Rewriter.* The Persistency Manager provides all the methods needed to store and manipulate the ontology components. In particular, this module of the system is able to deal with large distributed ontologies. Indeed, ontologies can be stored transparently in a number of text files and/or database management systems, possibly distributed across several machines.

Text files in OntoDLP format are analyzed by the *Parser* module that builds in main memory an image of the ontology components it recognizes; while, the *DB Manager* module is able to manipulate ontology entities that are imported into the system and stored in mass-memory by exploiting relational databases. In order to deal with virtual classes and virtual relations, we introduced a new submodule of the persistency manager, called *Virtual Entity Manager*, which will be described more in detail in the next subsection.

The *Persistency Manager* builds a global view of the distributed ontology which can be then exploited by the other components of the kernel, namely: *Type Checker* and *Rewriter.*

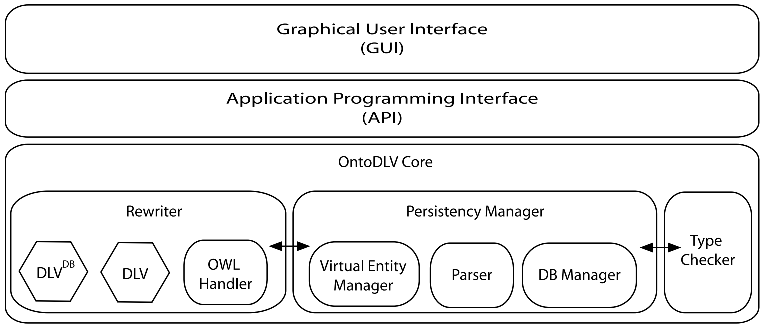

Fig. 2. The OntoDLV architecture.

The *Type Checker* module verifies the admissibility of an ontology by exploiting a number of type checking routines. It is important to say that, if the loaded ontology contains some admissibility problem (e.g., a class is declared twice) the type checker builds a precise description of the problem. This information can be exploited by external applications, and in particular the user interface of OntoDLV relies on this feature for helping the ontology design during the development process.

The *Rewriter* module translates OntoDLP ontologies, axioms, reasoning modules and queries to an equivalent ASP program [3] which, in the general case runs on the DLV system [9]. DLV is a state-of-the art ASP system that has been shown to perform efficiently on both hard and "easy" (having polynomial complexity) problems. Moreover, if the rewritten program is stratified and non disjunctive, then the evaluation is efficiently carried out on mass memory by exploiting a specialized version of DLV, called DLV^{DB} [**?**]. This feature has been purposely introduced to implement in an efficient way virtual classes and virtual relations. To this end, the original rewriter of OntoDLV has been extended to deal with the new features as described in the next subsection.

Finally, we recall that third parties are allowed for developing their own knowledge-based applications on top of OntoDLV by exploiting a rich Application Programming Interface: the OntoDLV API [13]. Moreover, the end user exploits the system through an easy-to-use and intuitive visual development environment called *GUI* (Graphical User Interface), which is built on top of the *OntoDLV API*. The OntoDLV GUI was designed to be simple for a novice to understand and use, and powerful enough to support experienced users.

4.2 Extension of the OntoDLV core: Virtual Entity Manager and New Rewriter

The implementation of *virtual classes* and *virtual relation* has been carried out by properly improving the *Rewriter* module and by adding a new submodule of the *Persistency Manager*, namely: the *Virtual Entity Manager*. The latter is in charge to make available suitable methods form defining, manipulating and storing both virtual classes and relations definitions.

More in detail, the *Virtual Entity Manager* implements two different usage modalities for virtual entities, that we call *off-line* and *on-line* modes. The first consists of materializing in OntoDLV the instances of virtual entities according with their respective mapping rules. Basically, a suitable routine performs the SQL queries on the proper database and each tuples of the result set is stored into the internal data structures (basically, instances are stored into the existing *DB manager* module). 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 materialization is obtained, the source database can be disconnected, since the data are stored into the OntoDLV persistency manager. Obviously, depending on database size, instance materialization could be time-consuming or even unpractical and, in addition, one may want keep the information in the database (which is accessed by legacy applications) in order to deal always with "updated" information. In this case the *on-line* mode is preferable, in which queries are performed directly at the sources. In fact, our implementation allows to efficiently perform reasoning/querying tasks directly on databases, with a very limited usage of main-memory. This can be achieved by exploiting DLV^{DB} [**?**], a specialized version of DLV that addresses the problem of reasoning on massive amounts of (possibly distributed) data.

In order to integrate DLV^{DB} in OntoDLV we extended the Rewriter module to generate the mapping statements that DLV^{DB} requires. In fact, to properly carry out the program evaluation, it is necessary to specify the mappings between input and output data and program predicates. For a better understanding, in the following we show which mappings are needed to rewrite the virtual class *branch*. We recall that the *branch* definition is:

virtual class *branch*(*name* : *string*, *city* : *string*, *assets* : *integer*){
  f(*BN*) : *branch*(*name* : *BN*, *city* : *BC*, *assets* : *A*) :−
    branch@*db*1(*branch-name* : *BN*, *branch-city* : *BC*, *assets* : *A*).}

Then the following directives for DLV^{DB} are generated by the new rewriter:

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

The above directive specifies the database source (**USEDB**) on which the SQL query will be performed. 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.

Note that, for obtaining the integration of a legacy database system we dealt with several non-trivial technical problems that are mainly due to the different ways in which different DBMSs store/represent data.[3] However, we refrain from reporting them here, mainly because of their inherent technical nature.

5 Conclusion and Related Work

In this paper we proposed a solution that allows one to "lift" an existing database 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 a large already-existent databases.

This has been obtained by properly extending the OntoDLV and system by means of *virtual classes* and *virtual relations*. The new modeling constructs allow the knowledge engineer for defining the instances of classes and relations of an enterprise ontology by means of special logic rules, which act as a mapping from the information stored in database tables to concept instances. In this way, the database is kept and the possibly already-existing applications can still work on it, but the user can take profit of the new ontological view of the data, and he/she can exploit the powerful reasoning mechanisms of the OntoDLV system for consistency checking, knowledge discovery, and other advanced knowledge-based tasks.

As a matter of fact, the problem of linking ontology to databases is not new [2] and has been studied also for other ontology languages. For instance, in [14] a set of pre-existing data sources is linked to the description logic DL-Lite$_{\mathcal{A}}$. In this approach, a very similar solution for creating object identifiers form database values (which exploits function symbols) 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). This makes the solution proposed in [14] very effective when dealing with large databases, and complexity-wise cheaper than our approach. However, the language of OntoDLV is much more expressive than DL-Lite$_{\mathcal{A}}$. Indeed, OntoDLP can express in a natural way more complex reasoning tasks on the ontology, which can be very useful for an enterprise, like e.g. solving an instance of the team building problem.[4]

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: PODS '02: Proceedings of the twenty-first ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems, New York, USA, ACM (2002) 233–246

[3] For example, there are several strings representation formats (quoted, unquoted, system reserved chars, etc.) or several different *date* types. Not all DBMS vendors adopt the same representation, so our routine must "understand" different formats and convert them properly according with the OntoDLV data-type system.

[4] Which amounts to finding a team of employees which satisfies some constraints on a project, like the overall budget, the maximum number of members, and so on.

3. Ricca, F., Leone, N.: Disjunctive Logic Programming with types and objects: The DLV$^+$ System. Journal of Applied Logics **5** (2007) 545–573
4. Dell'Armi, T., Gallucci, L., Leone, N., Ricca, F., Schindlauer, R.: OntoDLV: an ASP-based System for Enterprise Ontologies. In: Proceedings ASP07 - Answer Set Programming: Advances in Theory and Implementation. (2007)
5. Gelfond, M., Lifschitz, V.: Classical Negation in Logic Programs and Disjunctive Databases. NGC **9** (1991) 365–385
6. Gelfond, M., Leone, N.: Logic Programming and Knowledge Representation – the A-Prolog perspective . AI **138** (2002) 3–38
7. Minker, J.: Overview of Disjunctive Logic Programming. AMAI **12** (1994) 1–24
8. Eiter, T., Gottlob, G., Mannila, H.: Disjunctive Datalog. ACM TODS **22** (1997) 364–418
9. Leone, N., Pfeifer, G., Faber, W., Eiter, T., Gottlob, G., Perri, S., Scarcello, F.: The DLV System for Knowledge Representation and Reasoning. ACM TOCL **7** (2006) 499–562
10. Apt, K.R., Blair, H.A., Walker, A.: Towards a Theory of Declarative Knowledge. In: Foundations of Deductive Databases and Logic Programming. Washington DC (1988) 89–148
11. Przymusinski, T.C.: On the Declarative Semantics of Deductive Databases and Logic Programs. In: Foundations of Deductive Databases and Logic Programming. (1988) 193–216
12. Ullman, J.D.: Principles of Database and Knowledge Base Systems. Computer Science Press (1989)
13. Gallucci, L., Ricca, F.: Visual Querying and Application Programming Interface for an ASP-based Ontology Language. In: Proceedings of the Workshop on Software Engineering for Answer Set Programming (SEA'07). (2007) 56–70
14. Poggi, A., Lembo, D., Calvanese, D., Giacomo, G.D., Lenzerini, M., Rosati, R.: Linking Ontologies to Data. Journal on Data Semantics (2008) 133–173