To appear in: 8th International Conference on Data Warehousing and Knowledge Discovery (DaWak '06)

A Versioning Management Model for Ontology-Based Data Warehouses

Dung Nguyen Xuan[1] and Ladjel Bellatreche[1] and Guy Pierra[1]

LISI/ENSMA - Poitiers University
Futuroscope, FRANCE
E-mail : (nguyenx, bellatreche, pierra)@ensma.fr

Abstract. More and more integration systems use ontologies to solve the problem of semantic heterogeneities between autonomous databases. To automate the integration process, a number of these systems suppose the existence of a shared domain ontology a priori referenced by the local ontologies embedded in the various sources. When the shared ontology evolves over the time, the evolution may concern (i) the ontology level, (2) the local schema level, and/or (3) the contents of sources. Since sources are autonomous and may evolve independently, managing the evolution of the integrated system turns to an *asynchronous versioning problem*. In this paper, we propose an approach and a model to deal with this problem in the context of a materialized integration system. To manage the changes of contents and schemas of sources, we adapt the existing solutions proposed in traditional databases. To support ontology changes, we propose the *principle of ontological continuity*. It supposes that an evolution of an ontology should not make false an axiom that was previously true. This principle allows the management of each old instance using the new version of ontology. With this assumption, we propose an approach, called *the floating version model*, that fully automate the whole integration process. Our work is motivated by the automatic integration of catalogs of industrial components in engineering databases. It has been validated by a prototype using ECCO environment and the EXPRESS language.

1 Introduction

As digital repositories of information are springing up everywhere and interconnectivity between computers around the world is being established, the construction and evolution management of data warehouse over such *autonomous*, *heterogeneous* and distributed data sources becomes a crucial for a number of modern applications, such as e-commerce, concurrent engineering databases, integration systems, etc. A data warehouse can be seen as an integration system, where relevant data of various sources are extracted, transformed and materialized (contrary to the mediator architecture) in a warehouse [5]. To facilitate the construction of a data warehouse, two main issues may be considered: (1) the resolution of different conflicts (naming conflicts, scaling conflicts, confounding conflicts and representation conflicts) caused by semantic and schematic heterogeneities [8] and (2) the schematic autonomy of sources, known as the receiver heterogeneity problem [7]. To deal with the first issue, more and more approaches associated to data an ontology [18]. An ontology is defined as a formal specification of a shared conceptualization [9]. The main contribution of these ontologies is to formally represent the sense of instances of sources. In [3], we showed that when a shared (e.g., standardized) domain ontology exists, and each local source a priori references that ontology, an automatic integration becomes possible. Indeed, the articulation between the local ontologies and the shared one allows an automatic resolution of the different conflicts. Over the last years, a number of similar integration systems have been proposed following either mediator or warehouse architectures. In the mediator approach, we can cite, Picsel2 project for integrating web services, by using the OTA ontology (Open Travel Alliance) [19], and COIN project for exchanging financial data [8]. In the materialized approach, several ontology-based data management systems like RDFsuite [1] and DLDB [17] have been developed. The main assumption of these systems is that all the sources use the same shared ontology.

Most of the sources participating in the integration process operate autonomously, they are free to modify their ontologies and/or schemas, remove some data without any prior "public" notification, or occasionally block access to the source for maintenance or other purposes. Moreover, they may not always be aware of or concerned by other sources referencing them or integration systems accessing them [11, 21]. Consequently, the relation between the data warehouse and its sources is slightly coupled which causes anomalies of maintenance [6].

In the traditional databases, changes have two categories [20]: (1) content changes (insert/update/delete instances) and (2) schema changes (add/modify /drop attributes or tables). In order to tackle the problem of schema changes, two different ways are possible: *schema evolution* and *schema versioning*. The first approach consists in updating a schema and transforming data from an old schema into a new one (only the current version of a schema is present). In contrast, the second approach keeps track of the history of all versions of a schema. This approach is suitable for data warehousing environment where decision makers may need historical data [15]. In ontology-based integration systems, the evolution management is more difficult. This difficulty is due to the presence of ontologies (shared and local) that may also (slowly) evolve. In order to ensure the schematic autonomy of sources, some ontology-based integration systems allow also each source to *refine* the shared ontology by adding new concepts [3].

When the shared ontology evolves over the time and none global clock exits enforcing all the sources and the warehouse to evolve at the same time, various sources to be integrated may reference the same shared ontology as it was at various points in time. Therefore, the problem of integration turns to an asynchronous versioning problem. In this paper, we address this problem by considering the ontology-based integration system developed in our laboratory [3]. This system is based on three major assumptions that reflect our point of view on a large-scale integration: an automatic and a reliable integration of autonomous data sources is only possible if the source owners a priori agree on a common shared vocabulary. The only challenge is to define a mechanism that leaves as much as possible schematic autonomy of each source [3]. These assumptions are as follows: (1) Each data source participating in the integration process shall contain its own ontology. We call such a source an *ontology-based database* (OBDB) [3]. (2) Each local source a priori references a shared ontology by subsumption relationships "as much as possible" (i.e., each local class must reference its smallest subsuming class in the shared ontology). (3) A local ontology may restrict and extend the shared ontology as much as needed. The work proposed in this paper can be extended to others ontology-based integration systems. Although the evolution was largely studied [16], to the best of our knowledge, none of these systems considered the problem of asynchronous evolution of ontologies.

1.1 An Example of Changes in an Ontology-based Data Warehouse

Let's assume that we have a shared ontology representing Hard Disks. This ontology has one class having a set of properties (Interface, Transfer, Access time, Model, Capacity and Cache Memory). This ontology is referenced by a local ontology of source S_1. This ontology represents External Disks. It has also one class with a set of properties (Interface, Transfer, Model, Capacity, Codes). We assume that this class is stored in the source as a relational view. Initially, the shared and local ontologies have the version 1. Suppose that the shared and local ontologies evolve independently as follows: Concerning the shared ontology (1) the domain of the property "Interface" is extended, and (2) addition of a new property "Dimension". The source S_1 has the following changes: (1) the addition of a new property "Guaranteed" in the local ontology, (2) the renaming of the property "Codes" by "Series", (3) the deletion and the addition in the view of the source of the properties "Transfer" and "Guaranteed", respectively, and (4) the insertion/deletion of instances of the view.

In order to manage asynchronous evolution, the following issues need to be addressed: (1) the management of the evolutions of ontologies in order to maintain the relations between ontologies and the data originating from various sources [11], (2) the management of the life cycle of the instances (periods where an instance was alive), and (3) the capability to interpret each instance of the data warehouse, even if it is described using a different set of properties than those defined in the current version of the ontology (some properties are added/removed).

1.2 Contribution and Outline of the Paper

This paper is divided in six sections: Section 2 proposes our semantic integration approach based on a priori articulation between the shared ontology and local ontologies. Section 3 presents our approach to manage evolution of contents and schemas of data sources. Section 4 describes our mechanism of managing ontology changes using the principle of ontological continuity, and presents our floating version model. Section 5 presents an implementation of our approach using the Express language and ECCO environment. Section 6 concludes the paper by summarizing the main results and suggesting future work.

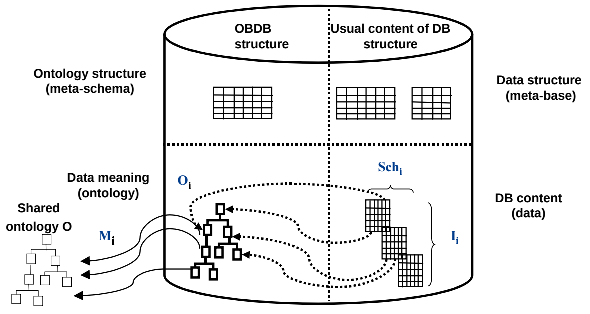


Fig. 1. The Structure of an Ontology-based Source

2 An a Priori Integration Approach

In this section, we formalize the ontology-based integration process in order to facilitate the presentation of our proposed solution. Let $S = \{S, ..., S_n\}$ be a set of sources participating in the integration process. Note that each source S_i has a local ontology O_i that references/extends the shared ontology O. Formally, the ontology O can be defined as the 4-tuples $< C, P, Applic, Sub >$, where:

- C is the set of the classes used to describe the concepts of a given domain,
- P is the set of all properties used to describe the instances of the classes of C. Note that only a subset of P might be selected by any particular database [1].
- *Applic* is a function defined as $Applic : C \rightarrow 2^P$. It associates to each class of the ontology, the properties that are rigid (applicable) for each instance of this class and that may be used, in the database, for describing its instances. Note that for each $c_i \in C$, only a subset of $Applic(c_i)$ may be used in any particular database, for describing c_i instances.
- *Sub* is the subsumption function defined as $Sub : C \rightarrow 2^C$ [2], where for a class c_i of the ontology, it associates its direct subsumed classes [3]. *Sub* defines a partial order over C. In our model, there exists two kinds of subsumption relationships: $Sub = OOSub \cup OntoSub$, where:
  - $OOSub$ is the usual object-oriented subsumption with the inheritance relationship. Through $OOSub$, applicable properties are inherited. $OOSub$ must define a single hierarchy.

[1] In our approach, each local ontology may also extend P.
[2] 2^C denotes the power set of C.
[3] C_1 subsumes C_2 iff $\forall x \in C_2, x \in C_1$.

- *OntoSub* is a subsumption relationship without the inheritance. Through *OntoSub* (also called case-of in the PLIB ontology model [18]), properties of a subsuming class may be imported by a subsumed class.
  OntoSub is also used as an *articulation operator* allowing to connect local ontologies into a shared ontology. Through this relationship, a local class may import or map all or any of the properties that are defined in the referenced class(es). In order to ensure the autonomy of sources, it may also define additional properties.

Now, we have all ingredients to define formally each source S_i as 5-tuples : $< O_i, Sch_i, I_i, Pop_i, M_i >$ (Figure 1), where:
(i) O_i is an ontology ($O_i :< C_i, P_i, Applic_i, Sub_i >$). (ii) $Sch_i : C_i \rightarrow 2^{P_i}$ associates to each ontology class $c_{i,j}$ of C_i the properties which are effectively used to describe the instances of the class $c_{i,j}$. This set may be any subset of $Appli(c_{ij})$ (as the role of an ontology is to conceptualize a domain, the role of a database schema is to select only those properties that are relevant for its target application). (iii) I_i is the set of instances of the source S_i. (iv) $Pop_i : C_i \rightarrow 2^{I_i}$ is the set of instances of each class. Finally, (v) the mapping M_i represents the *articulation* between the shared ontology O and the local ontology O_i. It is defined as a function: $M_i : C \rightarrow 2^{C_i}$, that defines the subsumption relationships without inheritance holding between C and C_i.

Several automatic integration scenarios may be defined in the above context [3]. For simplicity reason, we just outline below the *ExtendOnto* integration scenario, where the warehouse ontology consists of the shared ontology extended by the local ontologies of all the sources that have been added in the warehouse. Thanks to the articulation mappings (M_i), we note that all warehouse data that may be interpreted by the shared ontology (i.e., of which the class is subsumed by a shared ontology class) may be accessed through this ontology, whatever source they came from [4].

The ontology-based data warehouse DW has the same source structure (Figure 2): $DW :< O_{DW}, Sch_{DW}, I_{DW}, Pop_{DW}, \phi >$, where

1. O_{DW} is the warehouse ontology. It is computed by integrating local ontologies into the shared one. Its components are computed as follows:
   - $C_{DW} = C \cup (\cup_{1 \leq i \leq n} C_i)$.
   - $P_{DW} = P \cup (\cup_{1 \leq i \leq n} P_i)$.
   - $Applic_{DW}(c) = \begin{cases} Applic(c), \text{ if } c \in C \\ Applic_i(c), \text{ if } c \in C_i \end{cases}$
   - $Sub_{DW}(c) = \begin{cases} Sub(c) \cup M_i(c), \quad \text{if } c \in C \\ Sub_i(c), \text{ if } c \in C_i \end{cases}$
2. $I_{DW} = \cup_{1 \leq i \leq n} I_i$.
3. The instances are stored in tables as in their sources.
   - $\forall c_i \in C_{DW} \wedge c_i \in C_i (1 \leq i \leq n)$:

[4] another integration scenario, called *ProjOnto*, assumes that source instances are extracted after a projection operation on the shared ontology

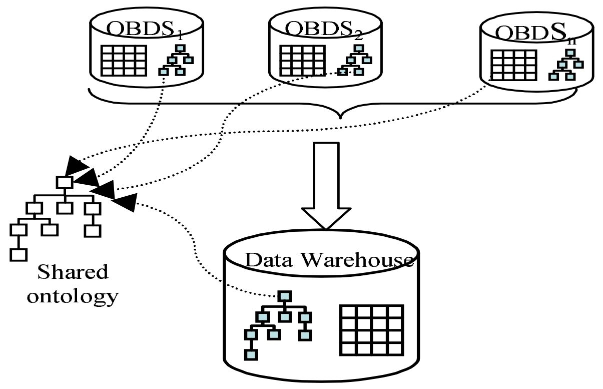


Fig. 2. The Structure of our Data Warehouse

(a) $Sch_{DW}(c_i) = Sch_i(c_i)$,
(b) $Pop_{DW}(c_i) = Pop_i(c_i)$

– $\forall c \in C$

(a) $Sch_{DW}(c) = Applic(c) \cap (Sch(c) \cup (\cup_{c_j \in Sub_{DW}(c)} Sch(c_j)))$.
(b) $Pop_{DW}(c) = \cup_{c_j \in Sub(c)} Pop(c_j)$

3 Evolution Management of Contents and Schemas

In this section, we present a mechanism to identify classes, properties and instances and the life cycle of instances.

To identify classes and properties, we use the universal identifiers (UI) defined in the ontology [3]. We assume that the identifiers contain two parts separated by ":". The first and second parts represent, an UI and a version number, respectively. In order to recognize instances of the data warehouse, any source must define for each class having a population a *semantic key*. It is composed by the representation (in character string form) of values of one or several applicable properties of this class.

3.1 The Life Cycle of Instances

In some situations, it may be useful to know the existence of instances in the warehouse at any previous point in time. To do so, we do not need to archive also the versions of ontologies since the current version always allows to interpret old instances (see Section 4). This problem is known by "schema versioning" [24], where all versioned data of a table are saved. Two solutions are possible to satisfy this requirement:

– In the *explicit storage* approach [2, 24], all the versions of each table are explicitly stored. This solution has two advantages: (i) it is easy to implement and allows an automation of the process of updating of data, and (ii) query processing is straightforward in cases where we precise the version on which the search will be done. On the other hand, the query processing cost can be very important if the query needs an exploration of all versioned data of the warehouse. Another drawback is due to the storage of the replicated data.

– In the *implicit storage* approach [24]: only one version of each table T is stored. This schema is obtained by making the *union* of all properties appearing in the various versions. On each data warehouse updating, one adds all existing instances of each source tables. Instances are supplemented by null values. This solution avoid the exploration of several versions of a given table. The major drawbacks of this solution are: (i) the problem of replicated data is still present, (ii) the implementation is more difficult than the previous one concerning the automatic computation of the schema of stored tables (the names of columns may have changed in the sources); (iii) the layout of the life cycle of data is difficult to implement (*"valid time"* [24]) and (iv) the semantics ambiguity of the null values.

Our solution follows the second approach and solves the problems in the following way:

1. the problem of *replicated data* is solved thanks to the single semantic identification (value of the semantic key) of each instance of data,
2. the problem of the updating process of table schemata is solved through the use of universal identifiers (UI) for all the properties.
3. the problem of the representation of the instances life cycle is solved by a pair of properties: $(Version_{min}, Version_{max})$. It enables us to know the validation period of a given instance.
4. the problem of the semantic ambiguity of the null values is handled by archiving the functions Sch of the various versions of each class. This archive enables us to determine the true schema of version of a table at any point in time, and thus the initial representation of each instance.

4 Ontology Evolution Management

4.1 Principle of Ontological Continuity

The constraints that may be defined in order to handle evolution of versioned ontology-based data sources result from the fundamental differences existing between the evolution of conceptual models and ontologies. A conceptual model is a model of a domain. This means, following the Minsky definition of a model [14], that it is an object allowing to respond to some particular questions on another object, namely, the target domain. When the questions change (when the organizational objectives of the target system are modified), its conceptual model is modified too, despite the fact that the target domain is in fact unchanged. Therefore, conceptual models are heavily depending upon the objectives assigned to the databases they are used to design. They evolve each time these objectives change. Contrary to conceptual models, an ontology is a conceptualization that is not linked with any particular objective of any particular computer system. It only aims to represent all the entities of a particular domain in a form that is consensual for a rather broad community having in mind a rather broad kind of problems. It is a logic theory of a part of the world, shared by a whole community, and allowing their members to understand each others. That can be, for example, the set theory (for mathematicians), mechanic (for mechanical engineers) or analytical counting (for accountants). For this type of conceptualizations, two changes may be identified: *normal evolution*, and *revolution*. A normal evolution of a theory is its deepening. New truths, more detailed are added to the old truths. But what was true yesterday remains true today. Concepts are never deleted contrary to [13].

It is also possible that axioms of a theory become false. In this case, it is not any more an evolution. It is a *revolution*, where two different logical systems will coexist or be opposed.

The ontologies that we are considered in our approach follow this philosophy. These ontologies are either standardized, for example at the international level, or defined by large size consortium which formalize in a stable way the knowledge of a technical domain. The changes in which we are interested are not those changes where all the shared knowledge of a domain is challenged by a new theory: we only address changes representing an *evolution* of the axioms of an ontology and not a *revolution*.

Therefore, we propose to impose to all manipulated ontologies (local and shared) to respect the following principle for ontology evolution:

Principle of ontological continuity: if we consider an ontology as a set of axioms, then ontology evolution must ensure that any true axiom for a certain version of an ontology will remain true for all its later versions. Changes that do not fulfill this requirement are called ontology revolution.

In the remaining paper, we only consider ontology evolution.

4.2 Constraints on the Evolution of Ontologies

In this section, we discuss the constraints for each kind of concept (classes, relation between classes, properties and instances) during ontology evolution. Let $O^k =< C^k, P^k, Sub^k, Applic^k >$ be the ontology in version k.

Permanence of the classes Existence of a class could not be denied across evolution: $C^k \subset C^{k+1}$. To make the model more flexible, as it is the case non computerized ontology, a class may become obsolete. It will then be marked as "deprecated", but it will continues belong to the newer versions of the ontology. In addition, the definition of a class could be refined, but this should not exclude any instance that was member of the class in the previous version. This means:

- the definition of a class may evolve,
- each class definition is to be associated with a version number.
- for any instance $i, i \in C^k \Rightarrow i \in C^{k+1}$.

Permanence of properties Similarly $P^k \subset P^{k+1}$. A property may become obsolete but neither its existence, nor its value for a particular instance may be modified. Similarly, a definition or value domain of a property may evolve. Taking into account the ontological principle of continuity, a value domain could be only increasing, certain values being eventually marked as obsolete.

Permanence of the Subsumption Subsumption is also an ontological concept which could not be infirmed. Let $Sub^* : C \rightarrow 2^C$ be the transitive closure of the direct subsumption relation Sub. We have then:

$$\forall\ C \in C^k, Sub^{*k}(c) \subset Sub^{*k+1}(c).$$

This constraint allows obviously an evolution of the subsumption hierarchy, for example by intercalating intermediate classes between two classes linked by a subsumption relation.

Description of instances The fact that a property $p \in Applic(c)$ means that this property is rigid [10] for each instance of c. This is an axiom that cannot be infirmed: $\forall c \in C^k, Applic^{*k}(c) \subset Applic^{*k+1}(c)$.

Note that this does not require that same properties are always used to describe the instances of the same class. As described in section 4.1, schematic evolution does not depend only on ontology evolutions. It depends also, and mainly, on the organizational objectives of each particular database version.

5 Floating Version Model: A Global Access to Current Instances

Before presenting our floating version model, we indicate the updating scenario of our data warehouse: at given moments, chosen by the data warehouse administrator, the current version of a source S_i is loaded in the warehouse. This version includes its local ontology, the mapping M_i between local ontology O_i and the shared ontology O, and its current content (*certain instances eventually already exist in the warehouse, others are new, others are removed*). This scenario is common in the engineering domain, where an engineering data warehouse consolidates descriptions (i.e., electronic catalogues) of industrial components of a whole of suppliers. Therefore, in this scenario, the maintenance process is carried out each time that a new version of an electronic catalogue of a supplier is received.

Our floating version model is able to support two kind of user services: (i) it allows to provide an access via a single ontology to the set of all instances that have been recorded in the data warehouse over the time its ontology and/or (ii) it also allows to record the various versions of the ontologies (shared and local) and to trace the life cycle of instances (full multi-version management). In this section we discuss how these objectives will be achieved.

The principal difficulty due to source autonomy is that in some situations, when two different sources are loaded, let's say S_i and S_j, a same class c of shared ontology O can be referred by an articulation mapping (i.e., subsumption) in different versions. For example, classes c_i^n of S_i and c_j^p of S_j may refer to c^k (class c with version k) and c^{k+j} (class c with version $k+j$), respectively. According to the principle of ontological continuity, it is advisable to note that:

1. all applicable properties in c^k are also applicable in c^{k+j},
2. all subsumed classes by c^k are also subsumed by c^{k+j},

Thus, the subsumption relation between c^k and c_i^n could be replaced by a subsumption relation between c^{k+j} and c_i^n. Moreover, all the properties that were imported from c^k may also be imported from c^{k+j}. Therefore, the class c^k is not necessary to reach (as a subsuming class) instances of c_i^n.

This remark leads us to propose a model, called the *floating version model*, which enables to reach all the data in the data warehouse via only one version of each class of the warehouse ontology. This set of versioned classes, called the *"current version"* of the warehouse ontology is such that the current version of each class c^f is higher or equal to the largest version of that class referenced by a subsumption relationship at the time of any maintenance. In practice, this condition is satisfied as follows:

- if an articulation M_i references a class c^f with a version lower than f, then M_i is updated in order to reference c^f,
- if an articulation M_i references a class c^f with a version greater than f, then the warehouse connect itself to the shared ontology server, loads the last version of the shared ontology and migrates all references M_i $(i = 1..n)$ to new current versions.

Example 1. During the maintenance process of a class C_1 ((Figure 3)) that references the shared ontology class C with version 2 **(1)**, the version of C in current ontology is 1 **(2)**. In this case, the warehouse downloads the current version of the shared ontology **(3)**. This one being 3, class C_1 is modified to reference version 3 **(4)**.

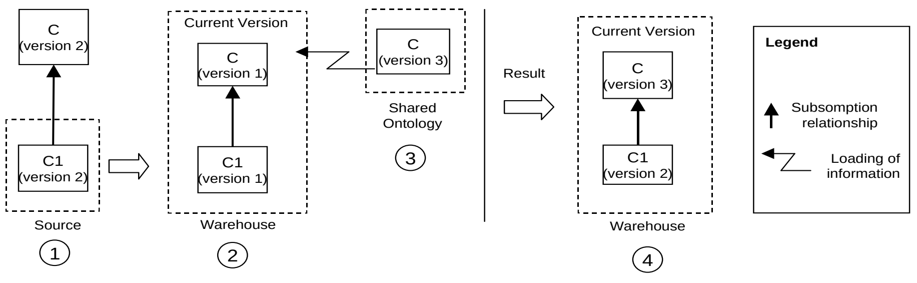


Fig. 3. A Model of the floating versions

We described below the two automatic maintenance processes that our floating version model makes possible.

5.1 Simplified Version Management

If the only requirements of users is to be able to browse the current instances of the data warehouse then, at each maintenance step: (1) ontology description of the various classes of the data warehouse ontology are possibly replaced by newer versions, and (2) the table associated to each class coming from a local ontology in the data warehouse is simply replaced by the corresponding current table in the local source.

5.2 A Full Multi-version Management

Note that in the previous case (Section 5.1), the articulation between a local ontology class and a shared ontology class stored in the current version of the data warehouse may not be its original definition (see the Figure 3). If the data warehouse user also wants to browse instances through the ontological definitions that existed when these instances were loaded, it is necessary to archive also all the versions of the warehouse ontology. This scenario may be useful, for example, to know the exact domain of an enumeration-valued property when the instance was defined. By implementing this possibility, we get a *multi-version data warehouse* which archives also all versions of classes having existed in the data warehouse life, and all the relations in their original forms. Note that the principle of ontological continuity seems to make seldom necessary this complex archive.

The multi-version data warehouse has three parts (see Figure 4):

1. *current ontology.* It contains the current version of the warehouse ontology. It represents also a generic data access interface to all instance data, whenever they were introduced in the warehouse.
2. *Ontology archive.* It contains all versions of each class and property of the warehouse ontology. This part gives to users the true definitions of versions of each concept. Versions of table schema T_i are also historized by archiving the function $Sch^k(c_i)$ of each version k of c_i, where T_i corresponds to the class c_i.
3. *multi-versioned tables.* It contains all instances and their version_min and version_max.

6 Implementation of our Approach

In order to validate our work, we have developed a prototype integrating several OBDSs (Figure 5), where ontologies and sources are described using the PLIB ontology model [18] specified by the Express language [22] . Such ontologies and instance data are exchangeable as instances of EXPRESS files (”physical file”).

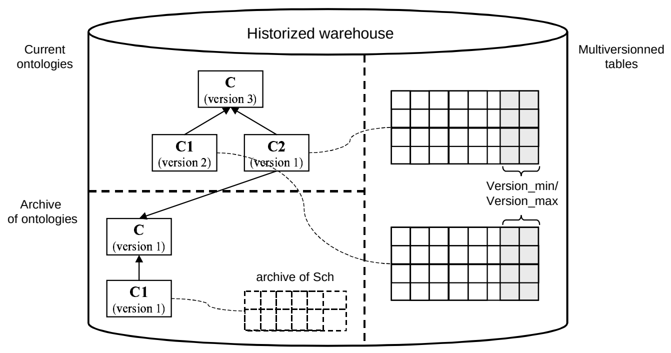


Fig. 4. Structure of warehouse integrated system

To process EXPRESS files, we used the ECCO Toolkit of PDTec which offers the following main functions [23]:

1. Edition, syntax and semantic checker of EXPRESS models;
2. Generation of functions (Java and C++) for reading, writing and checking integrity constraints of a physical file representing population of instances of an EXPRESS schema;
3. Manipulation of the population (physical file) of EXPRESS models using a graphical user interface;
4. Access to the description of a schema in the form of objects of a meta-model of EXPRESS;
5. Support of a programming language called EXPRESS-C. EXPRESS-C allows managing an Express schema and its instance objects.

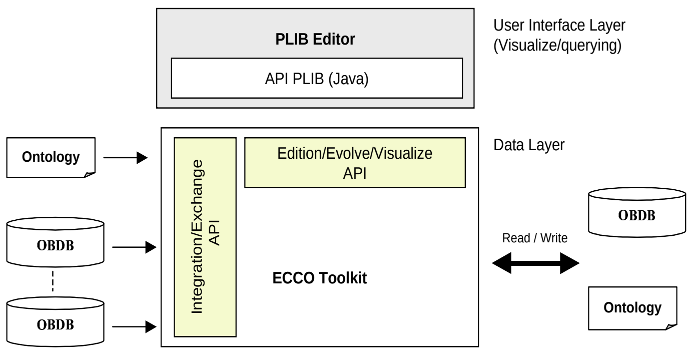
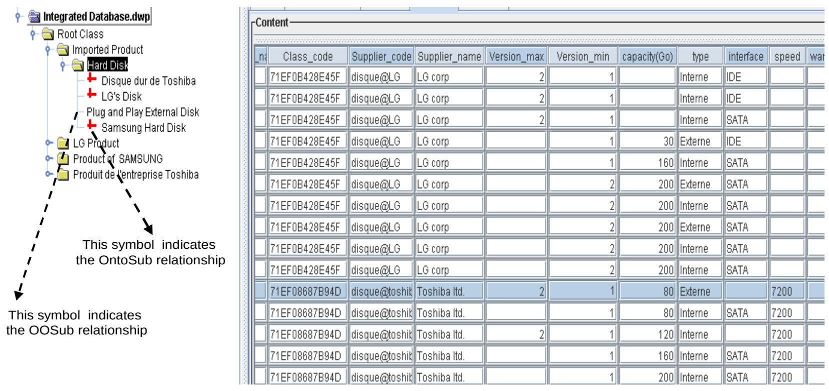


Fig. 5. Architecture of our Prototype

An ontology and an OBDS may be created via an editor called PLIBEditor. It is used also to visualize, edit and update ontologies (both shared and local) and sources. It uses a set of PLIB API developed under ECCO. PLIBEditor proposes a QBE-like graphical interface to query the data from the ontologies. This interface relies on the OntoQL query language [12] to retrieve the result of interactively constructed queries.

We have developed a set of integration API allowing the automatic integration both in the simplified version management scenario, and in the full multi-version management scenario.


Integrated Database.dwp
Root Class
Imported Product
Hard Disk
Disque dur de Toshiba
LG's Disk
Plug and Play External Disk
Samsung Hard Disk
LG Product
Product of SAMSUNG
Produit de l'entreprise Toshiba
This symbol indicates the OntoSub relationship
This symbol indicates the OOSub relationship
Content

_na	Class_code	Supplier_code	Supplier_name	Version_max	Version_min	capacity(Go)	type	interface	speed	war
	71EF0B428E45F	disque@LG	LG corp	2	1		Interne	IDE		
	71EF0B428E45F	disque@LG	LG corp	2	1		Interne	IDE		
	71EF0B428E45F	disque@LG	LG corp	2	1		Interne	SATA		
	71EF0B428E45F	disque@LG	LG corp		1	30	Externe	IDE		
	71EF0B428E45F	disque@LG	LG corp		1	160	Interne	SATA		
	71EF0B428E45F	disque@LG	LG corp		2	200	Externe	SATA		
	71EF0B428E45F	disque@LG	LG corp		2	200	Interne	SATA		
	71EF0B428E45F	disque@LG	LG corp		2	200	Externe	SATA		
	71EF0B428E45F	disque@LG	LG corp		2	200	Interne	SATA		
	71EF0B428E45F	disque@LG	LG corp		2	200	Interne	SATA		
	71EF08687B94D	disque@toshit	Toshiba ltd.	2	1	80	Externe		7200	
	71EF08687B94D	disque@toshit	Toshiba ltd.		1	80	Interne	SATA	7200	
	71EF08687B94D	disque@toshit	Toshiba ltd.	2	1	120	Interne		7200	
	71EF08687B94D	disque@toshit	Toshiba ltd.		1	160	Interne	SATA	7200	
	71EF08687B94D	disque@toshit	Toshiba ltd.		1	200	Interne	SATA	7200	

Fig. 6. Integrated hierarchical access and integrated querying over the data warehouse

Figure 6 shows the view offered to users over the content of the data warehouse after integration. The shared ontology (left side) provides for hierarchical access and query over the data warehouse content:

– a query over a shared ontology class allows to query all the classes subsumed either by the OOSub or by the OntoSub relationships, thus integrating instance data from all the integrated sources (see left side of Figure 6).
– hierarchical access allows also to go down until classes that came from any particular ontology (see right side of Figure 6).

7 Conclusion

In this paper, we presented the asynchronous versioning problem where autonomous ontology-based data sources are integrated in a ontology-based data warehouse. The sources that we considered are those containing local ontologies referencing in an a priori manner a shared one by subsumption relationships. These sources are autonomous and heterogeneous, and we assume that ontologies, schemas, and data may evolve over the time. Our integration process integrates first ontologies and then the data. The presence of ontologies allows an automation of the integration process, but it makes the management of autonomous sources more difficult.

Concerning ontology evolution, we described the difference between ontologies and database schemata and we suggested to distinguish ontology evolution and ontology revolution. Ontology evolution must respect the principle of ontological continuity that ensures that an axiom that was true for a particular version will remain true over all successive evolutions. This assumption allows the management of each old instance using a new version of the ontology.

Following this assumption, we have proposed two scenarios for the automatic integration of autonomous ontology-based data sources into an ontology-based data warehouse. Both scenarios are based on the floating version model, where the integration process always maintain a single version of the data warehouse ontology. This version, called the current ontology allows to interpret all the instance data in the data warehouse. The first scenario just updates the data warehouse ontology and data and provides automatically an integrated view of the current state of all the integrated sources. The data warehouse corresponding to the second scenario consists of three parts: (1) The current ontology contains the current version of the warehouse ontology. (2) The ontology archive contains all the versions of each class and property of the warehouse ontology. (3) The multi-versioned tables contain all instances and their first and last version of activities. This structure allows the tracing instances life cycle and the data access is done in a transparent manner. Note that for each source class, the set of properties identifying its instances is known, therefore it is possible to recognize the same instance when it is described by different properties. Our model was validated under ECCO by considering several local ontologies, where for each ontology, a set of sources instance data defined. This approach allows, in particular, an automatic integration of electronic component catalogues in engineering [4].

Concerning the perspectives, it would be interesting to consider (1) a mediator architecture of our proposed model and (2) the problem of view maintenance in our ontology-based warehouse.

References

1. S. Alexaki, V. Christophides, G. karvounarakis, D. Plexousakis, and K. Tolle. The ics-forth rdfsuite: Managing voluminous rdf description bases. *Proceedings of the Second International Workshop on the Semantic Web (SemWeb01)*, May 2001.
2. Bartosz Bebel, Johann Eder, Christian Koncilia, Tadeusz Morzy, and Robert Wrembel. Creation and management of versions in multiversion data warehouse. *Proceedings of the 2004 ACM symposium on Applied computing*, pages 717–723, June 2004.
3. L. Bellatreche, G. Pierra, D. Nguyen Xuan, H. Dehainsala, and Y. Ait Ameur. An a priori approach for automatic integration of heterogeneous and autonomous

databases. *International Conference on Database and Expert Systems Applications (DEXA'04)*, (475-485), September 2004.

4. L. Bellatreche, Xuan, G. Pierra, D. N., and H. Dehainsala. Contribution of ontology-based data modeling to automatic integration of electronic catalogues within engineering databases. *To appear in Computers in Industry Journal*, 2006.
5. S. Chaudhuri and U. Dayal. An overview of data warehousing and olap technology. *Sigmod Record*, 26(1):65–74, March 1997.
6. S. Chen, B. Liu, and E. A. Rundensteiner. Multiversion-based view maintenance over distributed data sources. *ACM Transactions on Database Systems*, 4(29):675–709, December 2004.
7. C. H. Goh, S. E. Madnick, and M. Siegel. Context interchange: Overcoming the challenges of large-scale interoperable database systems in a dynamic environment. *in Proceedings of the Third International Conference on Information and Knowledge Management (CIKM'94)*, pages 337–346, December 1994.
8. C.H. Goh, S. Bressan, E. Madnick, and M. D. Siegel. Context interchange: New features and formalisms for the intelligent integration of information. *ACM Transactions on Information Systems*, 17(3):270–293, 1999.
9. T. Gruber. A translation approach to portable ontology specification. *Knowledge Acquisition*, 5(2):199–220, 1995.
10. N. Guarino and C. A. Welty. Ontological analysis of taxonomic relationships. *in Proceedings of 19th International Conference on Conceptual Modeling (ER'00)*, pages 210–224, October 2000.
11. Jeff Heflin and James Hendler. Dynamic ontologies on the web. *Proceedings of the Seventeenth National Conference on Artificial Intelligence (AAAI/MIT Press)*, pages 443–449, 2000.
12. S. Jean, G. Pierra, and Y. Ait-Ameur. Ontoql: an exploitation language for obdbs. *VLDB Ph.D. Workshop*, pages 41–45, september 2005.
13. A. Maedche, B. Motik, L. Stojanovic, R. Studer, and R. Volz. Managing multiple ontologies and ontology evolution in ontologging. *Intelligent Information Processing*, pages 51–63, August 2002.
14. M. Minsky. Computer science and the representation of knowledge. *in The Computer Age: A Twenty-Year View, Michael Dertouzos and Joel Moses, MIT Press*, pages 392–421, 1979.
15. T. Morzy and R. Wrembel. Modeling a multiversion data warehouse : A formel approach. *International Conference on Entreprise Information Systems(ICESI'03)*, 2003.
16. Natalya F. Noy and Michel Klein. Semantic integration: a survey of ontology-based approaches. *SIGMOD Record*, 33(4), December 2004.
17. Z. Pan and J. Heflin. Dldb: Extending relational databases to support semantic web queries. Technical report, Dept. of Computer Science and Engineering, Lehigh University, USA, 2004.
18. G. Pierra, J. C. Potier, and E. Sardet. From digital libraries to electronic catalogues for engineering and manufacturing. *International Journal of Computer Applications in Technology (IJCAT)*, 18:27–42, 2003.
19. C. Reynaud and G. Giraldo. An application of the mediator approach to services over the web. *Special track "Data Integration in Engineering, Concurrent Engineering (CE'2003) - the vision for the Future Generation in Research and Applications*, pages 209–216, July 2003.
20. J. F. Roddick. A survey of schema versioning issues for database systems. *Information and Software Technology*, 37(7):383–393, 1995.

21. Elke A. Rundensteiner, Andreas Koealler, and Xin Zhang. Maintaining data warehouses over changing information sources. *Communications Of The ACM*, 43(6):57–62, June 2000.
22. D. Schenk and P. Wilson. *Information Modelling The EXPRESS Way*. Oxford University Press, 1994.
23. G. Staub and M. Maier. Ecco tool kit - an environnement for the evaluation of express models and the development of step based it applications. *User Manual*, 1997.
24. Han-Chieh Wei and Ramez Elmasri. Study and comparison of schema versioning and database conversion techniques for bi-temporal databases. *Proceedings of the Sixth International Workshop on Temporal Representation and Reasoning (IEEE Computer)*, May 1999.

This article was processed using the LaTeX macro package with LLNCS style