D2I

Integrazione, Warehousing e Mining di sorgenti eterogenee

Programma di ricerca (cofinanziato dal MURST, esercizio 2000)

A Generalized Schema Versioning Model for Object-Oriented and Semi-Structured Data

FABIO GRANDI

D1.R4 **30 aprile 2001**

Sommario

In this paper we introduce and describe $\mathcal{CVM}$, a generalized conceptual model for schema versioning support in a heterogeneous environment, where structured (object-oriented) and semi-structured data can be interoperated. The $\mathcal{CVM}$ model is aimed at representing (and reasoning on) both intensional and extensional aspects in a uniform way and consider "named" conceptual objects as first-class citizens. The $\mathcal{CVM}$ definition is based on a highly expressive and decidable Description Logic, $\mathcal{ALCQIO}$.

Tema	Tema 1: Applicazioni per basi di dati su Internet e Intranet
Codice	D1.R4
Data	30 aprile 2001
Tipo di prodotto	Rapporto tecnico
Numero di pagine	21
Unità responsabile	BO
Unità coinvolte	BO
Autore da contattare	Fabio Grandi Dipartimento di Elettronica, Informatica e Sistemistica Alma Mater Studiorum - Università di Bologna Viale Risorgimento 2, 40136 Bologna, Italia fgrandi@deis.unibo.it

A Generalized Schema Versioning Model for Object-Oriented and Semi-Structured Data

Fabio Grandi

30 aprile 2001

Abstract

In this paper we introduce and describe $\mathcal{CVM}$, a generalized conceptual model for schema versioning support in a heterogeneous environment, where structured (object-oriented) and semi-structured data can be interoperated. The $\mathcal{CVM}$ model is aimed at representing (and reasoning on) both intensional and extensional aspects in a uniform way and consider "named" conceptual objects as first-class citizens. The $\mathcal{CVM}$ definition is based on a highly expressive and decidable Description Logic, $\mathcal{ALCQIO}$.

1 Introduction

In this paper we define a generalized conceptual model for schema versioning support which can be used in a heterogeneous information integration setting. In particular, we introduce a conceptual model, which can account for object-oriented and semi-structured data, suitable to enable the adoption of a common export format from information sources for both data and metadata (e.g. based on XML [26]).

As a reference application environment we consider the same architecture for an integration system described, for instance, in [8, 9]. In particular, we follow the distinction between a *conceptual level*, with a global schema defined in an expressively rich language based on Description Logics [5], and a *logical level* where source schemata are represented. Owing to the definition of suitable mappings between the two levels, the global schema and the source schemata can then be reduced to a common rich formalism for the sake of reasoning and query processing. Within this framework, in this paper we are mainly interested in the conceptual level, as we will introduce a conceptual model able to capture the semantics of object-oriented and semi-structured data sources supporting schema versioning.

The conceptual model we will define, called $\mathcal{CVM}$ (Conceptual Versioning Model), is aimed at providing a uniform framework to represent (and reason on) both intensional and extensional aspects. In a sense, we would like to extend on a formal basis the simple mechanism adopted in relational systems, where data and metadata are represented in a uniform way as they are stored in base and catalog tables, respectively, although there is no support for mixed intensional/extensional reasoning at query language level. On the other hand, $\mathcal{CVM}$ represents a unifying framework to represent and query data and metadata in a multi-schema environment, with the support of a reasoning procedure for query containment (under the constraints imposed by a $\mathcal{CVM}$ schema) that we will show decidable. Query containment has been emphasized in several recent papers (e.g. [7, 21]) as a key problem for information integration (e.g. for query optimization or query rewriting using views).

In this context, we will consider as first-class citizens the conceptual objects which have an explicit denotation at user-interface level, that is the abstract entities which have a *name*, known by the users, which can be used in casual queries and compiled applications to reference data and which are also subject to change via schema modification. In our setting, such conceptual objects are classes and attributes (i.e. labeled record components). For these first-class citizens, we will distinguish between their intensional and extensional features, by means of reification into distinct concepts in Description Logics (DL). For instance, for each class C of objects with type T, we introduce an *intensional* concept c, which has the name C as a proper feature, and an *extensional* concept γ, representing the set of objects belonging to the class named C, having as a proper feature the conformance with the type T. Moreover, whereas the concept c maintains its identity across different schema versions, its features are subject to schema changes and also its extension γ can change as the same class can be populated by different objects in different schema versions. Intensional concepts will be represented as *nominals* [25], that is DL concepts representing a single individual.

The main motivation of this choice is the fact that in a real database (supporting schema versioning, in particular), the external names of schema objects are purely "accidental" with respect to the conceptual entities they represent indeed. In all the previous approaches where Description Logics have been used for useful modeling and reasoning at database schema level (e.g. in [11, 12]), names were *identified* with the conceptual objects they denote: a class (attribute) with name C (A) in the database was represented by a concept C (role A) in the DL. As a consequence, when schema changes are considered (e.g. in [16]), the change of a "surface property" as a class name is modeled via the creation of a new concept for the class with the new name, having the same extension as the concept standing for the class with the old name. Another important consequence is that, in such a way, it is impossible to reason on conceptual objects and their names in a uniform framework and, for example, ask intensional-extensional queries like:

(Q_1) Select the *current* values of the property
that was called A *in schema version* SV_1
carried by every object belonging *in schema version* SV_2
to the class named C *in schema version* SV_3

Furthermore, since we deal with an integration setting where structured object-oriented data have to be interoperated with semi-structured data, we will consider a rather "liberal" object model, non enforcing strict typing. Indeed, we will consider a conceptual model supporting object *polymorphism* (and without the so-called object/value dualism) in a fashion similar to that proposed in [6]. In this way, we aim at correctly modeling the "standard" attribute inheritance semantics between classes with record types and avoiding, at the same time, possible type conflicts and inconsistencies due to the (multiple) inheritance mechanism in a highly heterogenous integration environment. It should be noticed how, in this respect, the schema versioning support introduces an additional degree of heterogeneity also between the objects maintained in a single system, where the very same objects may be represented in different (even mutually inconsistent) ways. Note that in a system based on a *multi-pool* implementation solution [14], the same objects, in different schema versions, can also be associated to different values for the same attributes.

On the other hand, the $\mathcal{CVM}$ model will provide for quite rich expressiveness, by putting a full-boolean type language with record and set constructors at user's disposal. Basically, the single-schema data model we consider is very similar to the $\mathcal{CVL}$ object model proposed in [6]. However, we sacrifice part of its expressiveness concerning *roles* ($\mathcal{CVL}$ provides for a powerful sublanguage to express complex links between objects as regular expressions involving basic links) in order to maintain decidability also after the introduction of nominals. As a matter of fact, we only consider complex roles (in qualified number restrictions) involving inverse ($^-$) and union ($\sqcup$) constructors. However, since role union is not considered in standard $\mathcal{ALCQIO}$ (for which decidability and complexity characterization is available [25]), we will also discuss in Appendix A how its introduction does not impact on the $\mathcal{ALCQIO}$ computational properties, although it improves expressiveness.

$$
\begin{array}{rlrcl}
C, D \rightarrow & A \mid & & & \\
& \top \mid & \top^{\mathcal{I}} & = & \Delta^{\mathcal{I}} \\
& \bot \mid & \bot^{\mathcal{I}} & = & \emptyset \\
& \neg C \mid & (\neg C)^{\mathcal{I}} & = & \Delta^{\mathcal{I}} \setminus C^{\mathcal{I}} \\
& C \sqcap D \mid & (C \sqcap D)^{\mathcal{I}} & = & C^{\mathcal{I}} \cap D^{\mathcal{I}} \\
& C \sqcup D \mid & (C \sqcup D)^{\mathcal{I}} & = & C^{\mathcal{I}} \cup D^{\mathcal{I}} \\
& \forall R.C \mid & (\forall R.C)^{\mathcal{I}} & = & \{i \in \Delta^{\mathcal{I}} \mid \forall j.\ R^{\mathcal{I}}(i,j) \Rightarrow C^{\mathcal{I}}(j)\} \\
& \exists R.C \mid & (\exists R.C)^{\mathcal{I}} & = & \{i \in \Delta^{\mathcal{I}} \mid \exists j.\ R^{\mathcal{I}}(i,j) \wedge C^{\mathcal{I}}(j)\} \\
& \exists^{\geq n} R.C \mid & (\exists^{\geq n} R.C)^{\mathcal{I}} & = & \{i \in \Delta^{\mathcal{I}} \mid \sharp\{j \in \Delta^{\mathcal{I}} \mid R^{\mathcal{I}}(i,j) \wedge C^{\mathcal{I}}(j)\} \geq n\} \\
& \exists^{\leq n} R.C \mid & (\exists^{\leq n} R.C)^{\mathcal{I}} & = & \{i \in \Delta^{\mathcal{I}} \mid \sharp\{j \in \Delta^{\mathcal{I}} \mid R^{\mathcal{I}}(i,j) \wedge C^{\mathcal{I}}(j)\} \leq n\} \\
& o \mid & o^{\mathcal{I}} & = & \text{singleton subset of } \Delta^{\mathcal{I}} \\
& & & & \\
R, S \rightarrow & R \mid & & & \\
& R^{-} & (R^{-})^{\mathcal{I}} & = & \{(i,j) \in \Delta^{\mathcal{I}} \times \Delta^{\mathcal{I}} \mid R^{\mathcal{I}}(j,i)\} \mid \\
& R \sqcup S & (R \sqcup S)^{\mathcal{I}} & = & \{(i,j) \in \Delta^{\mathcal{I}} \times \Delta^{\mathcal{I}} \mid R^{\mathcal{I}}(i,j) \vee S^{\mathcal{I}}(i,j)\}
\end{array}
$$

Figure 1: $\mathcal{ALCQIO}$ (with role union) concept and role expressions and their semantics.

2 Preliminaries

We give here only a very brief introduction to the $\mathcal{ALCQIO}$ Description Logic; for a full account of $\mathcal{ALCQI}$ and $\mathcal{ALCQIO}$, see, for example [5, 25]. The basic types of a Description Logic are *concepts* and *roles*. The syntax rules at the left hand side of Figure 1 define valid concept and role expressions. Concepts are interpreted as sets of individuals—as for unary predicates—and roles as sets of pairs of individuals—as for binary predicates. Formally, an *interpretation* is a pair $\mathcal{I} = (\Delta^{\mathcal{I}}, \cdot^{\mathcal{I}})$ consisting of a set $\Delta^{\mathcal{I}}$ of individuals (the *domain* of $\mathcal{I}$) and a function $\cdot^{\mathcal{I}}$ (the *interpretation function* of $\mathcal{I}$) mapping every concept to a subset of $\Delta^{\mathcal{I}}$ and every role to a subset of $\Delta^{\mathcal{I}} \times \Delta^{\mathcal{I}}$, such that the equations at the right hand side of Figure 1 are satisfied.

A *knowledge base* is a finite set Σ of axioms of the form $C \sqsubseteq D$, involving concept expressions C, D; we write $C \equiv D$ as a shortcut for both $C \sqsubseteq D$ and $D \sqsubseteq C$. An interpretation $\mathcal{I}$ satisfies $C \sqsubseteq D$ if and only if the interpretation of C is included in the interpretation of D, i.e., $C^{\mathcal{I}} \subseteq D^{\mathcal{I}}$; it is said that C is subsumed by D. An interpretation $\mathcal{I}$ is a *model* of a knowledge base Σ iff every axiom of Σ is satisfied by $\mathcal{I}$. If Σ has a model, then it is *satisfiable*. Σ *logically implies* an axiom $C \sqsubseteq D$ (written $\Sigma \models C \sqsubseteq D$) if $C \sqsubseteq D$ is satisfied by every model of Σ. Reasoning in $\mathcal{ALCQIO}$ (i.e., deciding knowledge base satisfiability and logical implication) is decidable, and it has been proved to be a NExpTime-complete problem [25].

2.1 Classes and Attributes

As we anticipated in the Introduction, we consider Classes and Attributes as first-class citizens of $\mathcal{CVM}$, as they are entities *named* by users. Each of them is modeled, both at intensional and extensional levels, by means of *reification* into an individual concept.

In particular, every class $C : T$, with name C and whose instances are objects with type T (specified according to a given syntax), is basically represented in $\mathcal{CVM}$ by means of two concepts: c and γ, where c is a *nominal* representing the class at intensional level and γ represents its extension. The axioms defining the class $C : T$ in the conceptual schema are the following:

$$
\begin{array}{rcl}
\gamma & \equiv & \exists \mathsf{instance}^{-}.c \\
\gamma & \sqsubseteq & \psi(T)
\end{array}
$$

where $\psi(T)$ is the extension of type T in $\mathcal{CVM}$. The class object c is connected via a functional role name to a literal string representing its name (which can also be represented as a *nominal* C). A role instance connects the class concept c with all and only the objects in its extension γ.

Similarly, every attribute $A : T$, with name A and whose instances are objects carrying a value with type T, is represented by means of two concepts: a and α, where a is a *nominal* representing the attribute at intensional level and α represents its extension. The axioms defining the attribute $A : T$ in the conceptual schema are the following:

$$\alpha \equiv \forall \mathsf{instance}^{-}.a$$
$$\alpha \equiv \exists \mathsf{component}^{-}.\top \sqcap \exists \mathsf{value}.\psi(T)$$

where $\psi(T)$ is the extension of type T in $\mathcal{CVM}$. The attribute object a is connected via a functional role name to a literal string representing its name (which can also be represented as a *nominal* A). A role instance connects the attribute concept a with all and only the objects in its extension α (α contains an instance for every different record object that has the attribute a as a component). Notice that attribute extensions in a schema (version) are all disjoint, whereas class instances may share the same objects.

Moreover, the objects in the extension α are also connected by two other functional roles $\mathsf{component}^{-}$ and value to the (record) object whose the attribute is a component, and to their value (i.a. an object with type T), respectively. Notice that, in this way, a record type $T = [A_1 : T_1, \ldots, A_n : T_n]$ (where a_j and α_j are the concepts representing the A_j intension and extension) will be modeled as $\psi(T) \sqsubseteq \exists \mathsf{component}.(\alpha_1 \sqcap \exists \mathsf{value}.\psi(T_1)) \sqcap \cdots \sqcap \exists \mathsf{component}.(\alpha_n \sqcap \exists \mathsf{value}.\psi(T_n))$. Therefore, the attribute reification with (functional) roles $\mathsf{component}^{-}$ and value connecting to records and values is almost the same as the one introduced in [5] to substitute the binary relationship between record objects and attribute values (with $V_1 = \mathsf{component}^{-}$ and $V_2 = \mathsf{value}$). Moreover, in $\mathcal{CVM}$, the α concepts reify a ternary relationships indeed, which also involves the *intension* a, for which a third functional role $\mathsf{instance}^{-}$ is needed.

In a given database schema (schema version), intensional objects are also connected via a special functional role active to a nominal concept Yes or No, representing their "activation status". Once added (for the first time), new classes and attributes are created as *active*. Active classes and attributes can also be dropped via a successive schema change: in such a case they are not "physically" removed from the schema, they simply become *non active*. In this way, they (and their extensions) are not "forgotten" in the schema, and they can be successively *re-activated* by means of suitable schema changes. This is a common assumption in several schema versioning solutions (e.g. based on the *completed schema* notion [24, 15]), as the first requirement here is preserving as much information as possible, even in the presence of "destructive" schema changes, since also deleted information is always potentially amenable to be *reused* (e.g. when answering a *legacy* query). Therefore, the complete semantics of a class definition statement:

$$\underline{\text{Class}}\ C\ \underline{\text{type-is}}\ T$$

is the addition of the new concepts c, C (nominals) and γ to the knowledge base, plus the addition of the following terminological assertions involving the new individuals:

$$c \sqsubseteq \exists \mathsf{name}.C \sqcap \exists \mathsf{active}.\mathsf{Yes}$$

together with the terminological axioms involving c and γ as above to the TBox of the knowledge base associated with the current schema. Notice that such inclusion axiom implies the following assertions concerning individuals: $(c, C) : \mathsf{name}$ and $(c, \mathsf{Yes}) : \mathsf{active}$. Hence, by means of nominals, reasoning about schema individuals is reduced to TBox reasoning. Moreover, notice that in the presence of different schema versions, c is a "global" concept (i.e. defined once and valid in every schema version), whereas γ is a "local" concept, defined in a single schema version, since, in general, every class may have a *different extension* in each schema version.

Unlike classes, new attributes are not explicitly defined in "isolation" (i.e. by means of a dedicated statement) but are implicitly introduced through the definition of record types (in

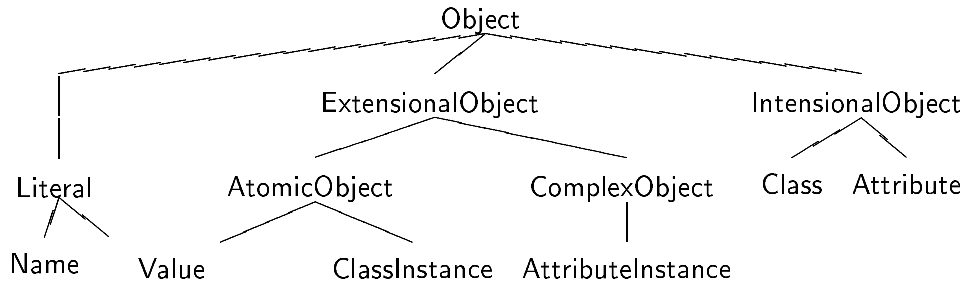

Figure 2: The $\mathcal{CVM}$ object hierarchy.

the type expression of a class declaration). For each attribute $A : T$ defined in a record type declaration, the new concepts a, A and α are introduced and the following assertions:

$$a \sqsubseteq \exists \mathsf{name}.A \sqcap \exists \mathsf{active}.\mathsf{Yes}$$

are added to the TBox with the axioms involving a and α as above. (ABox assertions (a, A) : name and (a, Yes) : active on schema individuals are implied). Notice that a is also a "global" concept, whereas α is "local" with respect to a specific schema version, since also attributes may have different extensions in different schema versions.

2.2 Objects

All the objects in the $\mathcal{CVM}$ domain comply with the hierarchy depicted in Fig. 2, that is they are organized according to the following taxonomy:

- IntensionalObject: the domain of *intensional objects*, denoting individual concepts of the schema, which can be referenced by means of a *name* at user-interface level; these can be of two types:
  - Class: the domain of objects representing a class defined in the schema
  - Attribute: the domain of objects representing an attribute defined (for a record in the type expression defined) for a class of the schema
- ExtensionalObject: the domain of *extensional objects*, which are used to represent data instances and link them with the intensional objects; these include:
  - AtomicObject: the domain of "visible" objects, representing data values as they can be manipulated at user-level; these can be of two types:
    - Value: the domain of "terminal" data values, that is literals representing a constant value of the domain (character data)
    - ClassInstance: the domain of objects belonging to the extension of a defined class, representing, when used as a "terminal", a reference to another individual object (like OIDs in object databases or "id/idref"s in XML)
  - ComplexObject: the domain of "hidden" objects, which are used to build the structure of complex types, linking individuals to their terminal data values; these include:
    - AttributeInstance: the domain of objects belonging to the extension of a defined attribute

General axioms ruling the $\mathcal{CVM}$ object hierarchy are:

$$
\begin{aligned}
\top &\equiv \mathsf{Object} \\
\mathsf{Object} &\equiv \mathsf{IntensionalObject} \sqcup \mathsf{ExtensionalObject} \sqcup \mathsf{Literal} \\
& \quad \mathsf{IntensionalObject} \sqsubseteq \neg\mathsf{ExtensionalObject} \\
& \quad \mathsf{Literal} \sqsubseteq \neg\mathsf{IntensionalObject} \sqcap \neg\mathsf{ExtensionalObject} \\
\mathsf{IntensionalObject} &\equiv \mathsf{Class} \sqcup \mathsf{Attribute} \\
& \quad \mathsf{Class} \sqsubseteq \neg\mathsf{Attribute} \\
\mathsf{ExtensionalObject} &\equiv \mathsf{AtomicObject} \sqcup \mathsf{ComplexObject} \\
& \quad \mathsf{AtomicObject} \sqsubseteq \neg\mathsf{ComplexObject} \\
\mathsf{AttributeInstance} &\sqsubseteq \mathsf{ComplexObject} \\
\mathsf{AtomicObject} &\equiv \mathsf{Value} \sqcup \mathsf{ClassInstance} \\
& \quad \mathsf{Value} \sqsubseteq \neg\mathsf{ClassInstance}
\end{aligned}
$$

where ClassInstance (AttributeInstance) is the set containing all the objects which can be instances of a class (attribute); attribute instances are all distinct.

If $\gamma_1, \ldots, \gamma_m$ $(\alpha_1, \ldots, \alpha_n)$ are the class (attribute) extensions in a given $\mathcal{CVM}$ schema (version), we have:

$$
\begin{aligned}
\gamma_1 \sqcup \cdots \sqcup \gamma_m &\sqsubseteq \mathsf{ClassInstance} \\
\alpha_1 \sqcup \cdots \sqcup \alpha_n &\sqsubseteq \mathsf{AttributeInstance} \\
& \quad \alpha_i \sqsubseteq \neg\alpha_j \qquad (1 \leq i < j \leq n)
\end{aligned}
$$

Name is the set of objects which are the extension of all distinguished nominals used as class and attribute names. Class and attribute names, as well as terminal attribute values are defined as character data literals; Literal also contains Yes and No distinguished nominals (and Yes and No actual values):

$$
\mathsf{Value} \sqcup \mathsf{Name} \sqcup \mathsf{Yes} \sqcup \mathsf{No} \sqsubseteq \mathsf{Literal}
$$

Definition 1 *A **Type/Data Graph** (*TDG*) in a given schema (version) is a* connected submodel *of $\mathcal{CVM}$, starting from a node in* Class*, having instances of* ComplexObject *as inner nodes and instances of* AtomicObject *as terminal nodes.*

In particular, each TDG is composed by a tree-shaped $\mathcal{CVM}$ submodel rooted on a node, say c, in Class intersecting (on common AttributeInstance nodes) all the tree-shaped $\mathcal{CVM}$ submodels rooted on a node in Attribute representing an attribute in the type of the class denoted by c (see Fig. 3). The leaves of the intersecting trees are the terminal nodes of the TDG.

Every $\mathcal{CVM}$ database is a collection of the TDGs corresponding to each defined class, having instances of AtomicObject as terminal nodes and instances of ComplexObject as inner nodes. Nodes in a TDG are linked by means of suitable $\mathcal{CVM}$ roles. These are:

- instance, linking each intensional object with the objects in its extension;
- name, connecting each intensional object to its name;
- active, telling whether the intensional object is active, or has been deleted, in the current database schema (resp. by connecting the object to the Yes or No nominal);

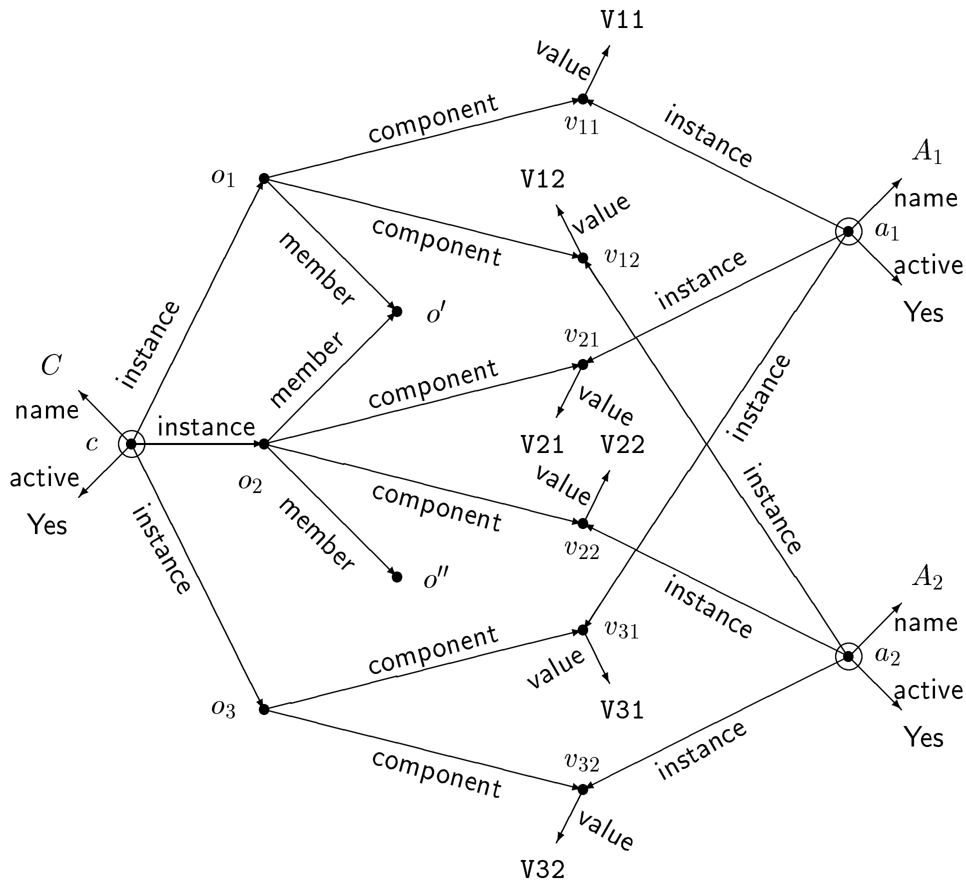

Figure 3: A sample TDG for the class denoted by c. Intensional objects c, a_1 and a_2 are evidenced with circles. Class and attribute extensions are: $\gamma = \{o_1, o_2, o_3\}, \alpha_1 = \{v_{11}, v_{21}, v_{31}\}, \alpha_2 = \{v_{12}, v_{22}, v_{32}\}$. Individual objects o' and o'' belong to the extension of another class (whose TDG is not drawn in figure).

- member, linking a set-type (complex) object with the objects belonging to the set;
- component, linking a record-type (complex) object to instances of the attributes defined for that record type;
- value, connecting an attribute instance with the terminal object representing its value;

In particular, a TDG contains directed edges labeled with instance, member, component and value role names. The constraints on which object pairs can be connected by each role can be expressed as follows:

$$\begin{array}{rcl}
\mathsf{IntensionalObject} & \sqsubseteq & \exists\mathsf{name}.\mathsf{Name} \sqcap \exists\mathsf{active}.(\mathsf{Yes} \sqcup \mathsf{No}) \\
\mathsf{Class} & \sqsubseteq & \exists\mathsf{instance}.\mathsf{ClassInstance} \\
\mathsf{Attribute} & \sqsubseteq & \exists\mathsf{instance}.\mathsf{AttributeInstance} \\
\mathsf{ComplexObject} \sqcup \mathsf{ClassInstance} & \sqsubseteq & \exists\mathsf{component}.\mathsf{AttributeInstance} \sqcup \exists\mathsf{member}.\mathsf{ExtensionalObject} \\
\mathsf{AttributeInstance} & \sqsubseteq & \exists\mathsf{value}.\mathsf{ExtensionalObject}
\end{array}$$

The following assertions enforce the functionality of the involved roles:

$$\begin{array}{rcl}
\top & \sqsubseteq & \exists^{\leq 1}\mathsf{name}.\top \sqcap \exists^{\leq 1}\mathsf{active}.\top \\
\mathsf{AttributeInstance} & \sqsubseteq & \exists^{\leq 1}\mathsf{instance}^{-}.\top \\
\top & \sqsubseteq & \exists^{\leq 1}\mathsf{component}^{-}.\top \sqcap \exists^{\leq 1}\mathsf{value}.\top \sqcap \exists^{\leq 1}\mathsf{member}^{-}.\top
\end{array}$$

Notice that the $\mathsf{instance}^{-}$ role is functional for attributes only, since attribute instances are all disjoint, whereas the same objects may belong to different class extensions (e.g. to represent *is-a* relationships). The further constraint:

$$\mathsf{ComplexObject} \quad \sqsubseteq \quad \exists^{\leq 1}(\mathsf{member}^{-} \sqcup \mathsf{component}^{-} \sqcup \mathsf{value}^{-}).\top$$

is very important for the semantics of $\mathcal{CVM}$, as it requires every *inner* node of a TDG to have exactly one predecessor object in every legal instance of the conceptual model. As a consequence, each inner node cannot be "reused" in the same or even in a different TDG: each edge in a TDG can only lead to a terminal object (AtomicObject) or to an always "fresh" inner-node object (ComplexObject). This means that any $\mathcal{CVM}$ knowledge base representing a legal database cannot contain *cycles* only involving objects in ComplexObject (i.e. cycles can only be closed through an individual in ClassInstance used as a terminal value of a TDG). Hence, any $\mathcal{CVM}$ model does not contain any "bad" cycle (in the sense of [12, Sec. 5.3]) corresponding, for example, to records with infinite depth or sets having themselves as members, which would not represent any meaningful database state. In other words, each TDG is *well-founded* in the sense of [6] thanks to the $\mathcal{CVM}$ structural constraints and does not require further specifications (or checks) to ensure it. Finally, we have:

$$\mathsf{Literal} \quad \sqsubseteq \quad \exists^{\leq 0}(\mathsf{instance} \sqcup \mathsf{name} \sqcup \mathsf{active} \sqcup \mathsf{member} \sqcup \mathsf{component} \sqcup \mathsf{value}).\top$$

stating that Literal objects are terminal nodes of TDGs.

Definition 2 *A* $\mathcal{CVM}$ Repository *is an* $\mathcal{ALCQIO}$ *knowledge base containing the definitions (intensional and extensional) of all the* TDG*s which are part thereof.*

In particular, every $\mathcal{CVM}$ Repository contains the specifications of a $\mathcal{CVM}$ *Schema*.

Definition 3 *A $\mathcal{CVM}$ Schema is a tuple $\mathcal{CVM} = (\mathsf{Object}, \mathcal{S}_0, \mathcal{SV}_\mathcal{S})$, where*

- Object *is a finite set of object instances;*
- $\mathcal{S}_0$ *is a knowledge base containing the basic $\mathcal{CVM}$ general axioms;*
- $\mathcal{SV}_\mathcal{S} = \{\mathcal{SV}_1, \ldots, \mathcal{SV}_s\}$ *is a finite set of knowledge bases, each of which contains the $\mathcal{CVM}$ axioms representing a schema version.*

Classes and attributes (denoted by intensional objects) may be active or not, may have different names and also different extensions in different schema versions. Therefore, each schema version $\mathcal{SV}_i$ contains a "private version" of every $\mathcal{CVM}$ role, that we will distinguish by means of numeric subscripts corresponding to the belonging schema version: name_i, active_i, $\mathsf{instance}_i$, member_i, $\mathsf{component}_i$ and value_i.

Notice that, whereas *basic* concepts are "global" in a $\mathcal{CVM}$ knowledge base, roles are "local" to a schema versions. For instance, we have as many $\mathsf{instance}_1, \ldots, \mathsf{instance}_s$ roles as many schema versions $\mathcal{SV}_1, \ldots, \mathcal{SV}_s$ (e.g. $\mathsf{instance}_i$ is the role connecting intensional objects to their instances in schema version $\mathcal{SV}_i$ and so on). Unlike basic concepts, class and attribute extensions (namely γ_is and α_js) are "versioned" in a $\mathcal{CVM}$ schema.

2.3 An Object-Oriented Syntax for Types

In this section we consider how $\mathcal{CVM}$ object types can be defined (e.g. at user's interface level) by means of a suitable syntax, which may correspond to some Object-Oriented data definition languages (e.g. as for the static part of ODMG).

In $\mathcal{CVM}$ we consider complex types built by means of Boolean operators, record and set constructors from a single basic atomic type CDATA (i.e. the same as in XML) for expressing character data, as usual for semi-structured information sources. Obviously, "traditional" strings and numbers can be encoded as CDATA literals and suitable operations to manipulate specific *subtypes* can be defined as class methods. However, the definition and management of methods is beyond the scope of this paper. Complex types can also be built from defined Classes to define, for example, ODMG-like relationships or *is-a* links implying (multiple) inheritance.

Attribute and class types can be defined as complex type expressions T built according to the following syntax:

$$
\begin{array}{rcl}
T & \rightarrow & C \ | \\
\text{(terminal type)} & & \mathsf{CDATA} \ | \\
\text{(complement type)} & & \underline{\mathsf{not}}\ T \ | \\
\text{(union type)} & & T_1\ \underline{\mathsf{or}}\ T_2 \ | \\
\text{(intersection type)} & & T_1\ \underline{\mathsf{and}}\ T_2 \ | \\
\text{(record type)} & & [\, A_1 : T_1, \ldots, A_k : T_k \,] \ | \\
\text{(set type)} & & \{T\}_{m:n} \ .
\end{array}
$$

where C and A_is are respectively class and attribute names, and $m : n$ denotes an optional constraint on the set cardinality.

For example, the TDG displayed in Fig. 3 represents a class c defined as:

$$\underline{\text{Class}}\ C\ \underline{\text{type-is}}\ [\, A_1 : \mathsf{CDATA}, A_2 : \mathsf{CDATA} \,]\ \underline{\mathsf{and}}\ \{C'\}$$

with instances:

$$
\begin{array}{rcl}
o_1 & : & [\, A_1 : \mathtt{V11}, A_2 : \mathtt{V12} \,]\ \underline{\mathsf{and}}\ \{o'\} \\
o_2 & : & [\, A_1 : \mathtt{V21}, A_2 : \mathtt{V22} \,]\ \underline{\mathsf{and}}\ \{o', o''\} \\
o_3 & : & [\, A_1 : \mathtt{V31}, A_2 : \mathtt{V32} \,]\ \underline{\mathsf{and}}\ \{\}
\end{array}
$$

The type constructors can be assigned a $\mathcal{CVM}$ semantics according to the following recursive rules:

$$\begin{array}{rcl}
\psi(C) & = & \exists\mathsf{instance}^{-}.(\exists\mathsf{name}.C) \\
\psi(\mathsf{CDATA}) & = & \mathsf{Literal} \\
\psi(\,\underline{\mathsf{not}}\ T) & = & \neg\psi(T) \\
\psi(T_1\,\underline{\mathsf{or}}\ T_2) & = & \psi(T_1)\sqcup\psi(T_2) \\
\psi(T_1\,\underline{\mathsf{and}}\ T_2) & = & \psi(T_1)\sqcap\psi(T_2) \\
\psi([\,A_1:T_1,\ldots,A_k:T_k\,]) & = & \exists^{=1}\mathsf{component}.\alpha_1\sqcap\cdots\sqcap\exists^{=1}\mathsf{component}.\alpha_n \\
\psi(\{T\}_{m:n}) & = & \forall\mathsf{member}.\psi(T)\sqcap\exists^{\geq m}\mathsf{member}.\top\sqcap\exists^{\leq n}\mathsf{member}.\top \quad .
\end{array}$$

Actually, the record type semantics has also, as a "side-effect", the addition of the terminological axioms ruling the new attributes to the knowledge base:

$$\begin{array}{rclrcl}
a_1 & \sqsubseteq & \exists\mathsf{name}.A_1 & a_1 & \sqsubseteq & \exists\mathsf{active}.\mathsf{Yes} \\
\alpha_1 & \equiv & \forall\mathsf{instance}^{-}.a_1 & \alpha_1 & \sqsubseteq & \exists\mathsf{component}^{-}.\top\sqcap\exists\mathsf{value}.\psi(T_1) \\
 & \cdots & & & \cdots & \\
a_n & \equiv & \exists\mathsf{name}.A_n & a_n & \sqsubseteq & \exists\mathsf{active}.\mathsf{Yes} \\
\alpha_n & \sqsubseteq & \forall\mathsf{instance}^{-}.a_n & \alpha_n & \sqsubseteq & \exists\mathsf{component}^{-}.\top\sqcap\exists\mathsf{value}.\psi(T_n)
\end{array}$$

where $a_1,\ldots,a_n$ are fresh nominals, $A_1,\ldots,A_n$ are nominals denoting the corresponding string objects in $\mathsf{Literal}$, $\alpha_1,\ldots,\alpha_n$ are the concepts representing the attribute extensions in a given schema (version).

Notice that the type system at user's disposal includes all the constructors usually needed to define semi-structured data. For example, syntax of *ssd-expressions* in [1] require OIDs, terminal values, and a labeled-record constructor. The availability of a set constructor allows the definition of collections of similar ssd-expressions (e.g. as it happens for XML data). Object polymorphism (as well as the availability of the type union constructor) is an additional "feature" that we consider very useful in a integration environment, where highly heterogeneous sources may be considered (e.g. HTML data or XML data non conforming to any DTD) together with structured sources, and the *same* objects may have very different representations in distributed information sources.

2.4 A Path Language for Attributes

Notice that, for every database schema (version), class names are global identifiers, whereas attribute names are unique identifiers only in the context of the (consistent) record type they are component of.

Due to uniqueness of (active) class names, the intensional class whose name is $C\in\mathsf{Name}$ can be denoted in $\mathcal{CVM}$ as:

$$c \ \sqsubseteq\ \mathsf{Class}\sqcap\exists\mathsf{name}.C\sqcap\exists\mathsf{active}.\mathsf{Yes}$$

Moreover, uniqueness of C can be checked by means of a reasoning task looking for *unsatisfiability* of a concept c' defined as follows:

$$c' \ \equiv\ \neg c\sqcap\mathsf{Class}\sqcap\exists\mathsf{name}.C\sqcap\exists\mathsf{active}.\mathsf{Yes}$$

Let us consider now denotation of attributes with respect to the type language, that will constitute a basic component of the external data manipulation and schema manipulation languages at user's disposal. As far as attributes are concerned, name uniqueness (and, consequently, consistency of record types) can be checked, for each attribute name $A\in\mathsf{Name}$, by means of a reasoning task consisting in *unsatisfiability* of a concept a' defined as follows:

$$\begin{array}{rcl}
a' & \equiv & \neg a\sqcap\mathsf{Attribute}\sqcap\exists\mathsf{name}.A\sqcap\exists\mathsf{active}.\mathsf{Yes}\sqcap\exists\mathsf{instance}.(\exists\mathsf{component}^{-}.o)\,, \qquad \text{where} \\
o & \sqsubseteq & \exists\mathsf{component}.(\exists\mathsf{instance}^{-}.(a\sqcap\mathsf{Attribute}\sqcap\exists\mathsf{name}.A\sqcap\exists\mathsf{active}.\mathsf{Yes}))
\end{array}$$

and a and o are fresh nominals. Since $a \sqcap \mathsf{Attribute} \sqcap \exists \mathsf{name}.A$ represents an (intensional) attribute a whose name is A, o denotes one record-type object having a as a component. Hence, a' is defined as an (intensional) attribute different from a with the same name A and being a component of the same record o. If such an a' exists, the record type whose o is an instance is inconsistent as it has two different attributes with the same name. Hence, if concept a' is unsatisfiable indeed, uniqueness of name A is ensured in the record type it belongs.

As far as attribute name uniqueness in the presence of record type built via Boolean constructors is concerned, we assume the following rules to be followed to obtain a "normalized" type definition with unrepeated attribute name occurrences in every record type:

$$\begin{array}{l} [\,A : T, A_1 : T_1, \ldots\,]\ \underline{\mathsf{and}}\ [\,A : T', A_2 : T_2, \ldots\,] \\ \quad := \quad [\,A : T\ \underline{\mathsf{and}}\ T'\,]\ \underline{\mathsf{and}}\ [\,A_1 : T_1, \ldots\,]\ \underline{\mathsf{and}}\ [\,A_2 : T_2, \ldots\,] \\ [\,A : T, A_1 : T_1, \ldots\,]\ \underline{\mathsf{or}}\ [\,A : T', A_2 : T_2, \ldots\,] \\ \quad := \quad [\,A : T\ \underline{\mathsf{or}}\ T'\,]\ \underline{\mathsf{and}}\ \big([\,A_1 : T_1, \ldots\,]\ \underline{\mathsf{or}}\ [\,A_2 : T_2, \ldots\,]\big) \end{array}$$

If attribute name uniqueness is enforced for each defined record type, individual attributes can still be uniquely referenced in a conceptual schema by means of a unique path expression, as described in the following. Although attributes are uniquely denoted by their name in the record they are component of, attributes with the same name can be freely used in the definition of different record types or even within the same complex type, as in the example which follows, representing a perfectly legal type structure:

$$T = [\,A_1 : [\,A_2 : T_1, A_3 : T_2\,], A_2 : [\,A_1 : T_3, A_3 : T_4\,]\,]$$

where each attribute name occurs twice but all six attributes are distinct and distinguishable. As a matter of fact, all the types in a schema can only be introduced as class type declarations and all class types have a tree structure rooted on the class itself. Therefore, it is always possible to uniquely identify each attribute in a schema by means of its name and the path which connects it to the class whose type it is part of. For instance, if class C has been declared with type T as in the above example, the attributes referenced by the leftmost (rightmost) occurrence of A_1 and A_3 can be denoted, respectively, as $C.A_1$ $(C.A_2.A_1)$ and $C.A_1.A_3$ $(C.A_2.A_3)$.

Taking into account also set constructors, every attribute in a schema can be uniquely denoted by a path expression with the form $X.A$ where the path X can be recursively built according to the following syntax:

$$X \rightarrow X.A \ \mid X \ni \mid C$$

For example, the path expression $C.A \ni B.A \ni\ni C.A$ represents the innermost A attribute of a class:

$$C : [\,A : \{[\,B : [\,A : \{\{[\,C : [\,A : T, \ldots\,], \ldots\,]\}\}, \ldots\,], \ldots\,]\}, \ldots\,]$$

Notice that, in general, one class and (possibly more than) one attribute may have the same name.

As to semantics, in the $\mathcal{CVM}$ conceptual model, the (active) intensional attribute referenced by the path expression $X.A$ can be uniquely denoted as:

$$a \equiv \mathsf{Attribute} \sqcap \exists \mathsf{name}.A \sqcap \exists \mathsf{active}.\mathsf{Yes} \sqcap \exists \mathsf{instance}.(\exists \mathsf{component}^{-}.(\varphi(X)))$$

where a is a fresh nominal and $\varphi(X)$ is recursively defined according to the following rules:

$$\begin{array}{rcl} \varphi(C) & = & \exists \mathsf{instance}^{-}.(\mathsf{Class} \sqcap \exists \mathsf{name}.C \sqcap \exists \mathsf{active}.\mathsf{Yes}) \\ \varphi(X \ni) & = & \exists \mathsf{member}^{-}.(\varphi(X)) \\ \varphi(X.A) & = & \exists \mathsf{value}^{-}.\big((\exists \mathsf{instance}^{-}.(\mathsf{Attribute} \sqcap \exists \mathsf{name}.A \sqcap \exists \mathsf{active}.\mathsf{Yes})) \sqcap (\exists \mathsf{component}^{-}.(\varphi(X)))\big) \end{array}$$

3 Reasoning Problems

According to the semantic definitions given in the previous section, several interesting reasoning problems can be introduced, in order to support the design and the management of a $\mathcal{CVM}$ repository with an evolving schema [16].

Definition 4 *Given a $\mathcal{CVM}$ repository with schema $\mathcal{S}$ we introduce the following reasoning problems:*

a. *Local/Global Schema Consistency: a $\mathcal{CVM}$ schema version $\mathcal{SV}_i$ of $\mathcal{S}$ is (locally) consistent if, as an $\mathcal{ALCQIO}$ knowledge base, $\mathcal{S}_0 \cup \mathcal{SV}_i$ is satisfiable (i.e. it admits a model); a $\mathcal{CVM}$ schema $\mathcal{S}$ is globally consistent if the $\mathcal{ALCQIO}$ knowledge base $\mathcal{S}_0 \cup \mathcal{SV}_1 \cup \cdots \cup \mathcal{SV}_s$ is satisfiable.*

b. *Local/Global Class Consistency: a $\mathcal{CVM}$ class named C is locally consistent in the schema version $\mathcal{SV}_i$ if there is at least one model $\mathcal{I}$ of $\mathcal{SV}_i$ such that the concept $\gamma_i \equiv \exists \mathsf{instance}_i^-.(\mathsf{Class} \sqcap \exists \mathsf{name}_i.C \sqcap \exists \mathsf{active}_i.\mathsf{Yes})$ is satisfiable, i.e. $\mathcal{S}_0 \cup \mathcal{SV}_i \not\models \gamma_i \sqsubseteq \perp$; a $\mathcal{CVM}$ class named C in $\mathcal{SV}_i$ is globally consistent in $\mathcal{S}$ if it is consistent in every schema version $\mathcal{SV}_j \in \mathcal{S}$, i.e. $\mathcal{S}_0 \cup \mathcal{SV}_1 \not\models \gamma_1 \sqsubseteq \perp, \ldots, \mathcal{S}_0 \cup \mathcal{SV}_s \not\models \gamma_s \sqsubseteq \perp$, where $\gamma_j \equiv \exists \mathsf{instance}_j^-.(\mathsf{Class} \sqcap \exists \mathsf{name}_i.C \sqcap \exists \mathsf{active}_j.\mathsf{Yes})$ for $j = 1..s$.*

c. *Local/Global Class Disjointness: two $\mathcal{CVM}$ classes named C, D are locally disjoint in the version $\mathcal{SV}_i$ if for every model $\mathcal{I}$ of $\mathcal{SV}_i$ the intersection of (the interpretation of) their extensions $\gamma_i \equiv \exists \mathsf{instance}_i^-.(\mathsf{Class} \sqcap \exists \mathsf{name}_i.C \sqcap \exists \mathsf{active}_i.\mathsf{Yes})$ and $\delta_i \equiv \exists \mathsf{instance}_i^-.(\mathsf{Class} \sqcap \exists \mathsf{name}_i.D \sqcap \exists \mathsf{active}_i.\mathsf{Yes})$ is empty, i.e. $\mathcal{S}_0 \cup \mathcal{SV}_i \models \gamma_i \sqcap \delta_i \sqsubseteq \perp$; two $\mathcal{CVM}$ classes, the former named C in $\mathcal{SV}_i$ and the latter D in $\mathcal{SV}_k$, are globally disjoint in $\mathcal{S}$ if for every model $\mathcal{I}$ of $\mathcal{S}_0 \cup \mathcal{SV}_1 \cup \cdots \cup \mathcal{SV}_s$, their extensions $\gamma_j \equiv \exists \mathsf{instance}_j^-.(\mathsf{Class} \sqcap \exists \mathsf{name}_i.C \sqcap \exists \mathsf{active}_j.\mathsf{Yes})$ and $\delta_j \equiv \exists \mathsf{instance}_j^-.(\mathsf{Class} \sqcap \exists \mathsf{name}_k.D \sqcap \exists \mathsf{active}_j.\mathsf{Yes})$ are disjoint in every schema version where they are both active, i.e. $\mathcal{S}_0 \cup \mathcal{SV}_1 \models \gamma_1 \sqcap \delta_1 \sqsubseteq \perp, \ldots, \mathcal{S}_0 \cup \mathcal{SV}_s \models \gamma_s \sqcap \delta_s \sqsubseteq \perp$.*

d. *Local/Global Class Subsumption: a class named D locally subsumes a class named C in the schema version $\mathcal{SV}_i$ if the extension $\gamma_i \equiv \exists \mathsf{instance}_i^-.(\mathsf{Class} \sqcap \exists \mathsf{name}_i.C \sqcap \exists \mathsf{active}_i.\mathsf{Yes})$ of C is included in the extension $\delta_i \equiv \exists \mathsf{instance}_i^-.(\mathsf{Class} \sqcap \exists \mathsf{name}_i.D \sqcap \exists \mathsf{active}_i.\mathsf{Yes})$ of D, i.e. $\mathcal{S}_0 \cup \mathcal{SV}_i \models \gamma_i \sqsubseteq \delta_i$; a class named D in $\mathcal{SV}_i$ globally subsumes a class named C in $\mathcal{SV}_k$ if $\mathcal{S}_0 \cup \mathcal{SV}_1 \models \gamma_1 \sqsubseteq \delta_1, \ldots, \mathcal{S}_0 \cup \mathcal{SV}_s \models \gamma_s \sqsubseteq \delta_s$, where $\gamma_j \equiv \exists \mathsf{instance}_j^-.(\mathsf{Class} \sqcap \exists \mathsf{name}_k.C \sqcap \exists \mathsf{active}_j.\mathsf{Yes})$ and $\delta_j \equiv \exists \mathsf{instance}_j^-.(\mathsf{Class} \sqcap \exists \mathsf{name}_i.D \sqcap \exists \mathsf{active}_j.\mathsf{Yes})$ for $j = 1..s$.*

4 Schema Changes

During the repository lifetime, a new schema version $\mathcal{SV}_j$ can be derived from an existing schema version $\mathcal{SV}_i$ via the application of the modification $\mathcal{M}_{ij}$. Each schema version $\mathcal{SV}_i \in \mathcal{SV}_\mathcal{S}$ has been derived in this way, starting from scratch ($\mathcal{S}_0$). In general, $\mathcal{M}_{ij}$ is a set of schema changes corresponding to the operators listed below. The schema change taxonomy is built by combining the schema elements which are subject to change with the elementary modifications (add, drop, reactivate and change) they undergo. Moreover, a <u>Merge-class</u> C, $\mathcal{SV}_k$, C' change is considered, which "merges" the definition of class C' in $\mathcal{SV}_k$ into the definition of class C in $\mathcal{SV}_i$ to produce the C class definition in $\mathcal{SV}_j$. The complete list is:

$M \rightarrow$ <u>Add-class</u> C, T | <u>Drop-class</u> C | <u>Reactivate-class</u> $\mathcal{SV}_k$, C |
<u>Change-class-name</u> C, C' | <u>Change-class-type</u> C, T' |
<u>Add-attribute</u> X.A, T | <u>Drop-attribute</u> X.A | <u>Reactivate-attribute</u> $\mathcal{SV}_k$, X.A |
<u>Change-attr-name</u> X.A, A' | <u>Change-attr-type</u> X.A, T' |
<u>Merge-class</u> C, $\mathcal{SV}_k$, C'

Notice also that the "reactivate"-type schema changes require the element to reactivate be non active in $\mathcal{SV}_i$ (i.e. it must have been dropped before or with the $\mathcal{SV}_i$ creation) but active in a schema version $\mathcal{SV}_k$, from which its definition is taken. Actually, $\mathcal{SV}_k$ is necessary to uniquely define the class to reactivate at intensional level. This is due to the fact that, whereas more than one non active class with the same name C can be present in $\mathcal{SV}_i$ (all distinct at intensional level), only one active class with the name C may be present in any schema version (including $\mathcal{SV}_k$).

An (operational) semantics of schema changes can be defined according to the specifications listed below. They all involve: (1) a reasoning task, checking for the legal applicability of the proposed schema change over the schema version $\mathcal{SV}_i$ under modification; (2) a "copy" from the knowledge base $\mathcal{SV}_i$ to the new knowledge base $\mathcal{SV}_j$ of all the axioms non affected by the schema change; (3) the completion of the $\mathcal{SV}_j$ knowledge base building by addition of the axioms concerning the modified part (e.g. axioms ruling a newly created class). Notice that the "copy" of axioms from $\mathcal{SV}_i$ to $\mathcal{SV}_j$ corresponds to an implicit inter-schema relationship between the two schema versions, as the corresponding data are forced to have the same (or similar) *type* structure. Therefore, a schema change enforces a set of implicit inter-schema constraints on the possible legal instances of the two schema versions but, in general, it does not imply any tighter constraint on the actual instances of the two schema versions, as they are allowed to evolve completely independently from each other: this happens, by definition, in a database supporting schema versioning under the multi-pool implementation solution [14] or in a heterogeneous environment where, for instance, spatial schema versions are used to encapsulate the distributed sources which are, in fact, absolutely independent [23].

Changes on Classes

Add-class C, T

1. check *unsatisfiability* of: $\mathsf{Class} \sqcap \exists \mathsf{name}_i.C \sqcap \exists \mathsf{active}_i.\mathsf{Yes}$ (if it fails, reject the schema change)
2. copy in $\mathcal{SV}_j$ all the assertions in $\mathcal{SV}_i$ (by changing the subscripts from i to j)
3. add to $\mathcal{SV}_j$ the axioms: $c \equiv \mathsf{Class} \sqcap \exists \mathsf{name}_j.C \sqcap \exists \mathsf{active}_j.\mathsf{Yes}$ and $\gamma_j \sqsubseteq \exists \mathsf{instance}_j^-.c \sqcap \psi_j(T)$, where c is a fresh nominal ($\psi_j(T)$ uses roles with subscript j and involves, if it (recursively) contains record type definitions, the addition of axioms defining the new attributes)

Drop-class C

1. check *satisfiability* of: $c \equiv \mathsf{Class} \sqcap \exists \mathsf{name}_i.C \sqcap \exists \mathsf{active}_i.\mathsf{Yes}$ (if it fails, reject the schema change)
2. copy in $\mathcal{SV}_j$ all the assertions in $\mathcal{SV}_i$ (by changing the subscripts from i to j) but the axioms involving c and its extension γ_i
3. add to $\mathcal{SV}_j$ the axiom: $c \equiv \mathsf{Class} \sqcap \exists \mathsf{name}_j.C \sqcap \exists \mathsf{active}_j.\mathsf{No}$

Reactivate-class $\mathcal{SV}_k$, C

1. check *satisfiability* of: $c \equiv \mathsf{Class} \sqcap \exists \mathsf{name}_k.C \sqcap \exists \mathsf{active}_k.\mathsf{Yes} \sqcap \exists \mathsf{name}_i.C \sqcap \exists \mathsf{active}_i.\mathsf{No}$ (if it fails, reject the schema change)
2. copy in $\mathcal{SV}_j$ all the assertions in $\mathcal{SV}_i$ (by changing the subscripts from i to j) and, from $\mathcal{SV}_k$ (by changing the subscripts from k to j), all the assertions involving the c extension γ_k
3. add to $\mathcal{SV}_j$ the axiom: $c \equiv \mathsf{Class} \sqcap \exists \mathsf{name}_j.C \sqcap \exists \mathsf{active}_j.\mathsf{Yes}$

Change-class-name C, C'

1. check *unsatisfiability* of: $\mathsf{Class} \sqcap \exists \mathsf{name}_i.C' \sqcap \exists \mathsf{active}_i.\mathsf{Yes}$ and *satisfiability* of: $c \equiv \mathsf{Class} \sqcap \exists \mathsf{name}_i.C \sqcap \exists \mathsf{active}_i.\mathsf{Yes}$ (if it fails, reject the schema change)
2. copy in $\mathcal{SV}_j$ all the assertions in $\mathcal{SV}_i$ (by changing the subscripts from i to j), including the axiom involving the c extension γ_i, but the axioms involving c as above
3. add to $\mathcal{SV}_j$ the axiom: $c \equiv \mathsf{Class} \sqcap \exists \mathsf{name}_j.C' \sqcap \exists \mathsf{active}_j.\mathsf{Yes}$

Change-class-type C, T'

1. check *satisfiability* of: $c \equiv \mathsf{Class} \sqcap \exists \mathsf{name}_i.C \sqcap \exists \mathsf{active}_i.\mathsf{Yes}$ (if it fails, reject the schema change)
2. copy in $\mathcal{SV}_j$ all the assertions in $\mathcal{SV}_i$ (by changing the subscripts from i to j) but $\gamma_i \sqsubseteq \exists \mathsf{instance}_i^-.c \sqcap \psi_i(T)$
3. add to $\mathcal{SV}_j$ the axiom: $\gamma_j \sqsubseteq \exists \mathsf{instance}_j^-.c \sqcap \psi_j(T')$
   ($\psi_j(T')$ uses roles with subscript j and involves, if it (recursively) contains record type definitions, the addition of axioms defining the new attributes)

Notice that other schema changes usually considered in the literature (e.g. [3, 16]) to directly manipulate the class type hierarchy can easily be effected through the Change-class-type operation. For instance, if class C has type T in $\mathcal{SV}_i$, the schema change Add-is-a C,C' can be effected as Change-class-type C, T and C'.

We consider now schema changes involving attributes. Notice that the involved attributes are denoted by means of the previously defined path language. This is a noteworthy extension, which provides for schema changes involving any attribute, even in the presence of multi-level nested record definitions, whereas schema evolution and versioning approaches (e.g. [16]) usually consider attribute modifications only for classes defined with a "flat" record type.

Changes on Attributes

Add-attribute X.A, T

1. check *unsatisfiability* of: $\mathsf{Attribute} \sqcap \exists \mathsf{name}_i.A \sqcap \exists \mathsf{active}_i.\mathsf{Yes} \sqcap \exists \mathsf{instance}_i.(\exists \mathsf{component}_i^-.\varphi_i(X))$ (if it fails, reject the schema change)
2. copy in $\mathcal{SV}_j$ all the assertions in $\mathcal{SV}_i$ (by changing the subscripts from i to j)
3. add to $\mathcal{SV}_j$ the axioms: $a \sqsubseteq \mathsf{Attribute} \sqcap \exists \mathsf{name}_j.A \sqcap \exists \mathsf{active}_j.\mathsf{Yes}$, $\alpha_j \equiv \forall \mathsf{instance}_j^-.a$ and $\alpha_j \sqsubseteq \exists \mathsf{component}_j^-.(\varphi_j(X)) \sqcap \exists \mathsf{value}_j.\psi_j(T)$. ($\varphi_j(X)$ and $\psi_j(T')$ use roles with subscript j and involve, if they (recursively) contain record type definitions, the addition of axioms defining also their attributes)

Drop-attribute X.A

1. check *satisfiability* of: $a \equiv \mathsf{Attribute} \sqcap \exists \mathsf{name}_i.A \sqcap \exists \mathsf{active}_i.\mathsf{Yes} \sqcap \exists \mathsf{instance}_i.(\exists \mathsf{component}_i^-.\varphi_i(X))$ (if it fails, reject the schema change)
2. copy in $\mathcal{SV}_j$ all the assertions in $\mathcal{SV}_i$ (by changing the subscripts from i to j) but the axiom involving a and its extension α_i
3. add to $\mathcal{SV}_j$ the axiom: $a \equiv \mathsf{Attribute} \sqcap \exists \mathsf{name}_j.A \sqcap \exists \mathsf{active}_j.\mathsf{No} \sqcap \exists \mathsf{instance}_j.(\exists \mathsf{component}_j^-.\varphi_j(X))$

Reactivate-attribute $\mathcal{SV}_k$, X.A

1. check and *satisfiability* of: $a \equiv \mathsf{Attribute} \sqcap \exists \mathsf{name}_k.A \sqcap \exists \mathsf{active}_k.\mathsf{Yes} \sqcap \exists \mathsf{instance}_k.(\exists \mathsf{component}_k^-.\varphi_k(X)) \sqcap$ $\sqcap \exists \mathsf{name}_i.A \sqcap \exists \mathsf{active}_i.\mathsf{No} \sqcap \exists \mathsf{instance}_i.(\exists \mathsf{component}_i^-.\varphi_i(X))$ (if it fails, reject the schema change)
2. copy in $\mathcal{SV}_j$ all the assertions in $\mathcal{SV}_i$ (by changing the subscripts from i to j) but the axiom involving a and, from $\mathcal{SV}_k$ (by changing the subscripts from k to j), all the assertions involving the a extension α_k
3. add to $\mathcal{SV}_j$ the axiom: $a \equiv \mathsf{Attribute} \sqcap \exists \mathsf{name}_j.A \sqcap \exists \mathsf{active}_j.\mathsf{No} \sqcap \exists \mathsf{instance}_j.(\exists \mathsf{component}_j^-.\varphi_j(X))$

Change-attr-name X.A, A'

1. check *satisfiability* of: $a \equiv \mathsf{Attribute} \sqcap \exists \mathsf{name}_i.A \sqcap \exists \mathsf{active}_i.\mathsf{Yes} \sqcap \exists \mathsf{instance}_i.(\exists \mathsf{component}_i^-.(\varphi_i(X) \sqcap$ $\neg \exists \mathsf{component}_i.\exists \mathsf{instance}_i^-.(\exists \mathsf{name}_i.A' \sqcap \exists \mathsf{active}_i.\mathsf{Yes})))$ (if it fails, reject the schema change)
2. copy in $\mathcal{SV}_j$ all the assertions in $\mathcal{SV}_i$ (by changing the subscripts from i to j), including the axiom involving the a extension α_i, but the axiom involving a as above
3. add to $\mathcal{SV}_j$ the axiom: $a \equiv \mathsf{Attribute} \sqcap \exists \mathsf{name}_j.A' \sqcap \exists \mathsf{active}_j.\mathsf{No} \sqcap \exists \mathsf{instance}_j.(\exists \mathsf{component}_j^-.\varphi_j(X))$

Change-attr-type X.A, T'

1. check *satisfiability* of: $a \equiv \mathsf{Attribute} \sqcap \exists \mathsf{name}_i.A \sqcap \exists \mathsf{active}_i.\mathsf{Yes} \sqcap \exists \mathsf{instance}_i.(\exists \mathsf{component}_i^-.(\varphi_i(X)))$ (if it fails, reject the schema change)
2. copy in $\mathcal{SV}_j$ all the assertions in $\mathcal{SV}_i$ (by changing the subscripts from i to j) but the axioms involving $\alpha_i \equiv \forall \mathsf{instance}_i^-.a$
3. add to $\mathcal{SV}_j$ the axioms: $\alpha_j \equiv \forall \mathsf{instance}_j^-.a$, $\alpha_j \sqsubseteq \exists \mathsf{component}_j^-.(\varphi_j(X)) \sqcap \exists \mathsf{value}_j.\psi_j(T)$

Notice that the use of nominals allow classes and attributes to preserve their *identity* across different schema versions regardless of renamings they possibly undergo. In previous approaches, where new concepts had to be introduced to cope with renaming, the relationship between the concept with the old name N and the one with the new name N' was usually modeled through the introduction of a *synonymity* between N and N'. However, such kind of synonymity is of a different kind (i.e. *stronger*) with respect to synonymity usually considered in an integration environment. The preservation of identity of renamed concepts avoids possible confusions between different kinds of synonymity, and allows us to deal with "standard" synonymities due to inter-schema constraints as usual, without the further complication of inter-version synonymities induced be renaming of conceptual objects in the schema.

We consider now the Merge-class schema change. It should be noticed that, at intensional level, such a primitive is sufficient to define all the merge-type schema changes usually considered in the literature (e.g. in [19]; notice that **AddProperty** and **PickProperty** of [19] are both equivalent, at intensional level, to an Add-attribute schema change: they only differ in the *change propagation*, as the former assigns null values to the new attribute and the latter adds a *populated* attribute by copying its values from a previous version.). The "classical" **MergeVersion** operation can be effected by repeated applications (for every present class) of the Merge-class primitive.

Merge-type changes

Merge-class C, $\mathcal{SV}_k$, C'

1. check *satisfiability* of: $c \equiv \mathsf{Class} \sqcap \exists \mathsf{name}_i.C \sqcap \exists \mathsf{active}_i.\mathsf{Yes}$ and: $c' \equiv \mathsf{Class} \sqcap \exists \mathsf{name}_k.C' \sqcap \exists \mathsf{active}_k.\mathsf{Yes}$ (if it fails, reject the schema change)
2. copy in $\mathcal{SV}_j$ all the assertions in $\mathcal{SV}_i$ (by changing the subscripts from i to j), including the axiom involving c as above, but the axiom involving the c extension γ_i
3. let the axioms concerning the c and c' extensions (resp. in $\mathcal{SV}_i$ and $\mathcal{SV}_k$) be $\gamma_i \sqsubseteq \exists \mathsf{instance}_i^-.c \sqcap \psi_i(T)$ and $\gamma_k \sqsubseteq \exists \mathsf{instance}_k^-.c' \sqcap \psi_i(T')$, then add to $\mathcal{SV}_j$ the axiom: $\gamma_j \sqsubseteq \exists \mathsf{instance}_j^-.c \sqcap (\psi_j(T) \sqcup \psi_j(T'))$ where $\psi_j(T)$ (resp. $\psi_j(T')$) is obtained by $\psi_i(T)$ (resp. $\psi_k(T')$) by turning all the role subscripts into j.

In general, a (complex) schema change is composed by a sequence of primitive schema changes followed by *change propagation* statements, which complete the schema change by rearranging the stored data at extensional level. Such statements, as the data to populate the new schema version may be computed as a query on the modified version(s), require the definition of a suitable query language to be used in a schema versioning environment.

5 Query Language

As far as the query language at user's disposal is concerned, we consider non-recursive Datalog queries, that is disjunctions of conjunctive queries, expressed in the general form:

$$q(\vec{\mathbf{x}}) \;\leftarrow\; body_1(\vec{\mathbf{x}}, \vec{\mathbf{y}}_1, \vec{\mathbf{c}}_1) \vee \cdots \vee body_q(\vec{\mathbf{x}}, \vec{\mathbf{y}}_q, \vec{\mathbf{c}}_q)$$

where each $body_i(\vec{\mathbf{x}}, \vec{\mathbf{y}}_i, \vec{\mathbf{c}}_i)$ is a conjunction of *atoms* and $\vec{\mathbf{x}}, \vec{\mathbf{y}}_i$ (resp. $\vec{\mathbf{c}}_i$) are all the variables (resp. constants) appearing in the conjunct. Each atom has the form $C(x)$ or $R(x, y)$ where

C (resp. R) is a primitive concept (resp. role), x and y are variables or constants in $\vec{\mathbf{x}}, \vec{\mathbf{y}}_i, \vec{\mathbf{c}}_i$. Constants can be thought as nominals denoting an individual in the domain. The number of variables in $\vec{\mathbf{x}}$ is the *arity* of the query q. Notice that roles in a query may belong to different $\mathcal{CVM}$ schema versions, allowing users to freely express the most general form of *multi-schema* queries. For instance, the (simple) example of multi-schema query Q_1 in the Introduction can be expressed (assuming $\mathcal{SV}_s$ be the *current* schema) as:

$$\begin{array}{rcl} q_1(x) & \leftarrow & \mathsf{Class}(y_1) \wedge \mathsf{name}_3(y_1, C) \wedge \mathsf{active}_3(y_1, \mathsf{Yes}) \wedge \\ & & \mathsf{active}_2(y_1, \mathsf{Yes}) \wedge \mathsf{instance}_2(y_1, y_2) \wedge \\ & & \mathsf{Attribute}(y_4) \wedge \mathsf{name}_1(y_4, A) \wedge \mathsf{active}_1(y_4, \mathsf{Yes}) \wedge \\ & & \mathsf{active}_s(y_4, \mathsf{Yes}) \wedge \mathsf{component}_s(y_2, y_3) \wedge \mathsf{instance}_s(y_4, y_3) \wedge \\ & & \mathsf{value}_s(y_3, x) \end{array}$$

Another interesting example of (intensional) multi-schema query is the following:

$$\begin{array}{rcl} q_2(x) & \leftarrow & \mathsf{Class}(y_1) \wedge \mathsf{name}_3(y_1, x) \wedge \\ & & \mathsf{active}_3(y_1, \mathsf{Yes}) \wedge \mathsf{active}_6(y_1, \mathsf{Yes}) \wedge \mathsf{active}_4(y_1, \mathsf{No}) \\ & \vee & \mathsf{Class}(y_2) \wedge \mathsf{name}_3(y_2, x) \wedge \\ & & \mathsf{active}_3(y_2, \mathsf{Yes}) \wedge \mathsf{active}_6(y_2, \mathsf{Yes}) \wedge \mathsf{active}_5(y_2, \mathsf{No}) \end{array}$$

asking for the names (in $\mathcal{SV}_3$) of all the classes that were active in $\mathcal{SV}_3$ and $\mathcal{SV}_6$ but had been dropped in between (i.e. they were not active in $\mathcal{SV}_4$ or $\mathcal{SV}_5$).

Given an interpretation $\mathcal{I}$ of a $\mathcal{CVM}$ schema $\mathcal{S}$, a query q for S of arity a is interpreted as the set $q^{\mathcal{I}}$ of a-tuples $(o_1, \ldots, o_a)$, with each $o_i \in \Delta^{\mathcal{I}}$, such that the FOL formula:

$$\exists \vec{\mathbf{y}}_1 .\, body_1(\vec{\mathbf{x}}, \vec{\mathbf{y}}_1, \vec{\mathbf{c}}_1) \vee \cdots \vee \exists \vec{\mathbf{y}}_q .\, body_q(\vec{\mathbf{x}}, \vec{\mathbf{y}}_q, \vec{\mathbf{c}}_q)$$

evaluates to true when substituting each o_i for x_i.

We consider now the *query containment* problem [7] (under the constraints imposed by a $\mathcal{CVM}$ schema), which is a central problem in several database applications, including data integration problems (e.g. [10, 21]). If q and q' are two queries of the same arity for $\mathcal{S}$, we say that q is *contained* in q' wrt $\mathcal{S}$ and write $\mathcal{S} \models q \subseteq q'$ iff $q^{\mathcal{I}} \subseteq (q')^{\mathcal{I}}$ for any model $\mathcal{I}$ of $\mathcal{S}$. Given a $\mathcal{CVM}$ schema $\mathcal{S}$ and two queries for $\mathcal{S}$:

$$\begin{array}{rcl} q(\vec{\mathbf{x}}) & \leftarrow & body_1(\vec{\mathbf{x}}, \vec{\mathbf{y}}_1, \vec{\mathbf{c}}_1) \vee \cdots \vee body_q(\vec{\mathbf{x}}, \vec{\mathbf{y}}_q, \vec{\mathbf{c}}_q) \\ q'(\vec{\mathbf{x}}) & \leftarrow & body'_1(\vec{\mathbf{x}}, \vec{\mathbf{y}}'_1, \vec{\mathbf{c}}'_1) \vee \cdots \vee body'_{q'}(\vec{\mathbf{x}}, \vec{\mathbf{y}}'_{q'}, \vec{\mathbf{c}}'_{q'}) \end{array}$$

we have that $\mathcal{S} \models q \subseteq q'$ iff there is no model of $\mathcal{S}$ that makes true the formula:

$$\begin{array}{l} \big(body_1(\vec{\mathbf{x}}, \vec{\mathbf{y}}_1, \vec{\mathbf{c}}_1) \vee \cdots \vee body_q(\vec{\mathbf{x}}, \vec{\mathbf{y}}_q, \vec{\mathbf{c}}_q)\big) \wedge \\ \neg\exists \vec{\mathbf{z}}_1 .\, body'_1(\vec{\mathbf{x}}, \vec{\mathbf{z}}_1, \vec{\mathbf{c}}'_1) \wedge \cdots \wedge \neg\exists \vec{\mathbf{z}}_{q'} .\, body'_{q'}(\vec{\mathbf{x}}, \vec{\mathbf{z}}_{q'}, \vec{\mathbf{c}}'_{q'}) \end{array}$$

The *query containment problem* consists of checking whether $\mathcal{S} \models q \subseteq q'$ for assigned $\mathcal{S}, q, q'$. In $\mathcal{CVM}$ it can be solved as a reasoning task consisting in checking *inconsistency* of an $\mathcal{ALCQI}$-T_CBox [25] $\mathcal{T}$, that is a set of cardinality constraints on $\mathcal{ALCQI}$ concept expressions in the form $(\leqslant n.\ C)$ or $(\geqslant n.\ C)$. In order to define the required T_CBox, we slightly "augment" our $\mathcal{CVM}$ knowledge base to include a new nominal ω (denoting a *starting point* which we add to the domain) and a newly defined role U through which each individual in the domain can be reached from $\omega^{\mathcal{I}}$ (including $\omega^{\mathcal{I}}$ itself; actually $U^- \circ U$ represents the *universal role*):

$$\begin{array}{rcl} (\Delta')^{\mathcal{I}} & = & \Delta^{\mathcal{I}} \cup \{\omega^{\mathcal{I}}\} \\ \mathcal{S}' & = & \mathcal{S} \cup \{\top \sqsubseteq \exists^{=1} U^-.\top, \omega \equiv \exists U.\top\} \end{array}$$

Hence $\mathcal{T}$ can be defined as:

$$\mathcal{T} \quad = \quad T_C(\mathcal{S}') \cup T_C(var) \cup T_C(q) \cup T_C(q') \cup T_C(\omega)$$

where:

$$\begin{array}{rcl}
T_C(\mathcal{S}') & = & \bigcup_{\{C \sqsubseteq D\} \in \mathcal{S}'} \{(\leqslant 0.\ C \sqcap \neg D)\} \cup \bigcup_{o \text{ nominal in } \mathcal{S}'} \{(\leqslant 1.\ o), (\geqslant 1.\ o)\} \\
T_C(var) & = & \bigcup_{v \text{ in } (\vec{\mathbf{x}}, \vec{\mathbf{y}}_i, \vec{\mathbf{c}}_i, \vec{\mathbf{c}}'_j)} \{(\leqslant 1.\ A_v), (\geqslant 1.\ A_v)\} \cup \bigcup_{v \text{ in } (\vec{\mathbf{z}}_j)} \{(\geqslant 0.\ A_v)\} \\
T_C(q) & = & \{(\geqslant 1.\ \bigsqcup_{i=1..q} \Big(\big(\bigsqcap_{C(v) \text{ in } body_i} \exists U.(A_v \sqcap C) \big) \sqcap \big(\bigsqcap_{R(u,v) \text{ in } body_i} \exists U.(A_u \sqcap \exists R.A_v) \big) \Big))\} \\
T_C(q') & = & \{(\geqslant 1.\ \bigsqcap_{j=1..q'} \Phi_{body'_j})\} \\
T_C(\omega) & = & \bigcup_{\{C \sqsubseteq D\} \in \mathcal{S}'} \{(\geqslant 1.\ \omega \sqcap \neg C)\} \cup \bigcup_{v \text{ in } (\vec{\mathbf{x}}, \vec{\mathbf{y}}_i, \vec{\mathbf{c}}_i, \vec{\mathbf{c}}'_j, \vec{\mathbf{z}}_j)} \{(\geqslant 1.\ \omega \sqcap \neg A_v)\}
\end{array}$$

where $\Phi_{body'_j}$ represents the encoding of each $\neg \exists \vec{\mathbf{z}}_j.\, body'_j(\vec{\mathbf{x}}, \vec{\mathbf{z}}_j, \vec{\mathbf{c}}'_j)$ and can be built (in a similar way as in [7]) as follows. We start from a *dependency-graph* (similar to the tuple-graph in [7]) which evidences the *cyclic* dependencies between variables [13]. The dependency-graph is a directed graph with nodes labeled by $\mathcal{ALCQI}$ concepts and edges labeled by roles defined as:

- there is one node v for each term in $\vec{\mathbf{x}}, \vec{\mathbf{z}}_j, \vec{\mathbf{c}}'_j$, labeled by A_v and by all C such that atom $C(v)$ appears in $body'_j$;
- there is one edge from node u to node v, labeled by R, for each atom $R(u, v)$ occurring in $body'_j$.

The dependency-graph is, in general, composed of $m_j \geq 1$ connected components. For the ℓ-th component, we build an $\mathcal{ALCQI}$ concept $\Delta_\ell(\vec{\mathbf{z}}_j)$, starting from a node v_0 and visiting the component as follows. Let u the current node in the visit and ϕ_u the formula produced by visiting u; if u has already been visited, then $\phi_u = A_u$, otherwise is it the intersection of:

- every concept labeling the node u (including A_u);
- $\exists R.(A_v \sqcap \phi_v)$ for each non-marked edge (u, v) labeled by R, where ϕ_v is the concept resulting by marking the edge (u, v) and visiting the node v;
- $\exists R^-.(A_v \sqcap \phi_v)$ for each non-marked edge (v, u) labeled by R, where ϕ_v is the concept resulting by marking the edge (v, u) and visiting the node v.

Then $\Delta_\ell(\vec{\mathbf{z}}_j) = \phi_{v_0}$ and $\Phi_{body'_j}$ is obtained as the conjunction of all concepts obtained by replacing in

$$\bigsqcup_{\ell=1..m_j} \forall U. \neg \Delta_\ell(\vec{\mathbf{z}}_j)$$

each concept A_v, with v in $\vec{\mathbf{z}}_j$, occurring in a *cycle* in the dependency-graph by each of the concepts A_u corresponding to a variable (or constant) u in $(\vec{\mathbf{x}}, \vec{\mathbf{y}}_i, \vec{\mathbf{c}}_i, \vec{\mathbf{c}}'_j)$. The number of such conjuncts in $\Phi_{body'_j}$ is $O(\ell_1^{\ell_2})$, where ℓ_1 is the number of variables and constants in q plus the number of constants in q', and ℓ_2 is the number of z variables occurring in a *cycle* of the dependency graph for q'.

Lemma 1 *Let $\mathcal{S}'$ be an augmented $\mathcal{CVM}$ schema and q, q' two queries as above. Then deciding whether $\mathcal{S} \models q \subseteq q'$ can be done in* NExpTime *by checking* inconsistency *of the $\mathcal{ALCQI}$-T_CBox $\mathcal{T}$.*

Proof (sketch). The correctness of such encoding can be proved, taking into account that the interpretation of the newly introduced concepts A_v's represent in $T_C(q)$ and $T_C(q')$ values in the range of the corresponding variable v. The first part of $T_C(var)$ states that all the variables in $(\vec{\mathbf{x}}, \vec{\mathbf{y}}_i, \vec{\mathbf{c}}_i, \vec{\mathbf{c}}'_j)$ assume as value an individual of the domain, whereas the second part of $T_C(var)$ states that the variables in $(\vec{\mathbf{z}}_j)$ may assume any value in the domain. The substitutions made in $T_C(q')$ for the z variables occurring in cycles of the dependency graphs accounts for the fact that the $body'_j$ may evaluate to true or false due to the assignments to variables and constants in $(\vec{\mathbf{x}}, \vec{\mathbf{y}}_i, \vec{\mathbf{c}}_i, \vec{\mathbf{c}}'_j)$ forced by q and other conjuncts in q' and to the dependency between variables (see also [7]). The $T_C(\omega)$ constraints only ensure that the newly added object $\omega^{\mathcal{I}}$ does not "interfer" with satisfiability of other constraints on Δ; in particular, it can be proved that $\mathcal{S}$ is satisfiable iff $\mathcal{S}'$ is satisfiable and $\mathcal{S} \models q \subseteq q'$ iff $\mathcal{S}' \models q \subseteq q'$.

Completeness of the encoding ($\mathcal{T}$ inconsistent $\Rightarrow \mathcal{S}' \not\models q \subseteq q'$): The consistent T_CBox $\mathcal{T}$ admits a model $\mathcal{I}$ that we can check makes true q and not q'. First of all, $\mathcal{I}$ satisfies all the constraints in $T_C(\mathcal{S}')$ and, thus, is a model of $\mathcal{S}'$. The models of $\mathcal{S}'$ are *connected* [2] (due to the existence of the universal role) and every individual of $\mathcal{I}$ can be reached from $\omega^{\mathcal{I}}$. In particular, $T_C(q)$ consistency states that from $\omega^{\mathcal{I}}$ we can reach a tuple of objects that makes true q and is compatible with the assignments of domain individuals to variables and constants ensured by the first part of $T_C(var)$. On the other hand, $T_C(q')$ consistency states that there is no combined assignment of variables and constants in q and any choice of the $(\vec{\mathbf{z}}_j)$ values in the domain that makes true any $body'_j$ conjunct.

Soundness of the encoding ($\mathcal{T}$ consistent $\Rightarrow \mathcal{S}' \models q \subseteq q'$): With a similar reasoning, one can verify that every model $\mathcal{I}$ of $\mathcal{S}'$ in which there is at least one tuple satisfying q and not q' satisfies all the constraints in the T_CBox $\mathcal{T}$.

Complexity: Consistency of an $\mathcal{ALCQI}$-T_CBox is a NExpTime-complete problem [25]. □

It can easily be shown that the same results still hold if atoms in the $body_i$ conjuncts are also allowed to include equality predicates between the involved variables and constants. On the other hand, we conjecture that decidability is lost if inequalities are allowed in query conjuncts (as it happens for the conceptual model based on $\mathcal{DLR}_{reg}$ studied in [7]). An intuition on this fact can be given by considering that the introduction of inequalities corresponds to provide for a *transitive* role to mimick the order relation $\leq$ in the $\mathcal{ALCQIO}$ Description Logic. Moreover, it has been shown (e.g. in [20]) that the addition of transitive roles in the presence of a role hierarchy in a strict sublanguage of $\mathcal{ALCQIO}$ leads to undecidability (the availability of the role union constructor easily allows to define a role hierarchy since $R \sqsubseteq R \sqcup S$, see also [18]).

Other useful reasoning tasks involving queries in an integration context, like *query consistency* and *query disjointness*, can be reduced to deciding query containment as shown in [10].

References

[1] S. Abiteboul, P. Buneman, and D. Suciu. *Data on the Web: From Relations to Semistructured Data and XML*. Morgan Kaufmann Publishers, San Francisco, CA, 1999.

[2] F. Baader. Augmenting Concept Languages by Transitive Closure of Roles: an ALternative to Terminological Cycles. In *Proc. of Intl' Joint Conf. on Artificial Intelligence (IJCAI'91)*, 1991.

[3] J. Banerjee, W. Kim, H.-J. Kim, and H. F. Korth. Semantics and Implementation of Schema Evolution in Object-Oriented Databases. In *Proc. of the ACM-SIGMOD Annual Conf.*, pages 311–322, May 1987.

[4] A. Borgida. On the Relative Expressiveness of Description Logics and First Order Logics. *Artificial Intelligence*, 82:353–367, 1996.

[5] D. Calvanese, G. De Giacomo, M. Lenzerini, and D. Nardi. Reasoning in Expressive Description Logics. In A. Robinson and A. Voronkov, editors, *Handbook of Automated Reasoning*. Elsevier, 2000.

[6] D. Calvanese, G. De Giacomo, and M. Lenzerini. Structured Objects: Modeling and Reasoning. In *Proc. Intl' Conf. on Deductive and Object-oriented Databases (DOOD'95)*, 1995.

[7] D. Calvanese, G. De Giacomo, and M. Lenzerini. On the Decidability of Query Containment under Constraints. In *Proc. of the 17th ACM SIGACT SIGMOD SIGART Symposium on Principles of Database Systems (PODS'98)*, pages 149–158, 1998.

[8] D. Calvanese, G. De Giacomo, M. Lenzerini, D. Nardi, and R. Rosati. Description Logic Framework for Information Integration. In *Proc. of the 6th Intl' Conf. on the Principles of Knowledge Representation and Reasoning (KR'98)*, pages 2–13, 1998.

[9] D. Calvanese, G. De Giacomo, M. Lenzerini, D. Nardi, and R. Rosati. Information Integration: Conceptual Modeling and Reasoning Support. In *Proc. of Intl' Conf. on Cooperative Information Systems (CoopIS'98)*, 1998.

[10] D. Calvanese, G. De Giacomo, M. Lenzerini, D. Nardi, and R. Rosati. Schema and Data Integration Methodology for DWQ. Technical Report DWQ-UNIROMA-004, Foundations of Data Warehouse Quality (DWQ), 1998.

[11] D. Calvanese, M. Lenzerini, and D. Nardi. Description Logics for Conceptual Data Modeling. In J. Chomicki and G. Saake, editors, *Logics for Databases and Information Systems*, pages 229–263. Kluwer Academic Publishers, 1998.

[12] D. Calvanese, M. Lenzerini, and D. Nardi. Unifying Class-based Representation Formalisms. *Journal of Artificial Intelligence Research*, 11:199–240, 1999.

[13] C. Chekuri and A. Rajaraman. Conjunctive query containment revisited. In *Proc. of Intl. Conf. on Database Theory (ICDT'97)*, LNCS 1186, Delphi, Greece, 1997.

[14] C. De Castro, F. Grandi, and M. R. Scalas. Schema Versioning for Multitemporal Relational Databases. *Information Systems*, 22(5):249–290, 1997.

[15] R. T. Snodgrass (ed.), I. Ahn, G. Ariav, D. Batory, J. Clifford, C. E. Dyreson, R. Elmasri, F. Grandi, C. S. Jensen, W. Käfer, N. Kline, K. Kulkarni, T. Y. Cliff Leung, N. Lorentzos, J. F. Roddick, A. Segev, M. D. Soo, and S. M. Sripada. *The TSQL2 Temporal Query Language*. Kluwer Academic Publishers, 1995.

[16] E. Franconi, F. Grandi, and F. Mandreoli. A Semantic Approach for Schema Evolution and Versioning in Object-Oriented Databases. In *Proc. Intl' Conf. on Deductive and Object-Oriented Databases (DOOD 2000)*, 2000.

[17] E. Grädel, M. Otto, and E. Rosen. Two-variable Logic with Counting is Decidable. In *Proc. 12th Annual IEEE Symposium on Logic in Computer Science*, 1997.

[18] F. Grandi. On Expressive Number Restrictions in Description Logics. In *Proc. of the Intl' Workshop on Description Logics (DL'01)*. Preliminary version available via anonymous ftp as `ftp://ftp-db.deis.unibo.it/pub/fabio/TR/CSITE-07-01.pdf`, 2001.

[19] F. Grandi, F. Mandreoli, and M. R. Scalas. A Generalized Modeling Framework for Schema Versioning Support. In *Proc. of 11th Australasian Database Conf. (ADC 2000)*, January 2000.

[20] I. Horrocks, U. Sattler, and S. Tobies. Practical Reasoning for Very Expressive Description Logics. *Logics Journal of the IGPL*, 3(3):239–263, 2000.

[21] T. Millstein, A. Levy, and M. Friedman. Query Containment for Data Integration Systems. In *Proc. of the 19th ACM SIGACT SIGMOD SIGART Symposium on Principles of Database Systems (PODS'00)*, pages 67–75, 2000.

[22] L. Pacholski, W. Szwast, and L. Tendera. Complexity of Two-variable Logic with Counting. In *Proc. 12th Annual IEEE Symposium on Logic in Computer Science*, 1997.

[23] J. F. Roddick, F. Grandi, F. Mandreoli, and M. R. Scalas. Beyond Schema Versioning: a Flexible Model for Spatio-Temporal Schema Selection. *Geoinformatica*, 5(1):33–50, March 2001.

[24] J. F. Roddick and R. T. Snodgrass. Schema Versioning. In *The TSQL2 Temporal Query Language*, pages 427–449. Kluwer Academic Publishers, 1995.

[25] S. Tobies. The Complexity of Reasoning with Cardinality Restrictions and Nominals in Expressive Description Logics. *Journal of Artificial Intelligence Research*, 12, 2000.

[26] The Extensible Markup Language (XML) Home Page, W3C Consortium. `http://www.w3.org/XML/`.

A Complex Roles for Free in $\mathcal{ALCQIO}$

As a matter of fact, the Description Logic we consider here should be better named $\mathcal{ALCQIOB}$, that is $\mathcal{ALCQIO}$ extended with Boolean constructors (e.g. union, intersection, ...) on atomic roles, that is extended with the constructs defined below:

$$
\begin{array}{lllcl}
R, S \rightarrow & R \sqcup S & (R \sqcup S)^{\mathcal{I}} & = & \{(i,j) \in \Delta^{\mathcal{I}} \times \Delta^{\mathcal{I}} \mid R^{\mathcal{I}}(i,j) \vee S^{\mathcal{I}}(i,j)\} \\
& R \sqcap S & (R \sqcap S)^{\mathcal{I}} & = & \{(i,j) \in \Delta^{\mathcal{I}} \times \Delta^{\mathcal{I}} \mid R^{\mathcal{I}}(i,j) \wedge S^{\mathcal{I}}(i,j)\} \\
& \neg R & (\neg R)^{\mathcal{I}} & = & \Delta^{\mathcal{I}} \times \Delta^{\mathcal{I}} \setminus R^{\mathcal{I}} \\
& R \Rightarrow S & (R \Rightarrow S)^{\mathcal{I}} & = & \{(i,j) \in \Delta^{\mathcal{I}} \times \Delta^{\mathcal{I}} \mid \neg R^{\mathcal{I}}(i,j) \vee S^{\mathcal{I}}(i,j)\} \\
& R \setminus S & (R \setminus S)^{\mathcal{I}} & = & \{(i,j) \in \Delta^{\mathcal{I}} \times \Delta^{\mathcal{I}} \mid R^{\mathcal{I}}(i,j) \wedge \neg S^{\mathcal{I}}(i,j)\}
\end{array}
$$

We show that the addition of the new constructs does not change the decidability of the Logics which remains in NExpTime. As a matter of fact, concept satisfiability, concept subsumption and knowledge base satisfiability in $\mathcal{ALCQIO}$ can be reduced to checking the consistency of an $\mathcal{ALCQI}$-T$_C$Box [25], by substituting (as we did in Sec. 5) every nominal o with a "normal" $\mathcal{ALCQI}$ concept O with additional cardinality restrictions $\{(\leqslant 1\, O), (\geqslant 1\, O)\}$ in order to force its interpretation to be a singleton. Obviously, starting from an extended $\mathcal{ALCQIO}$ knowledge bases, the new role constructs will also be present in the resulting extended $\mathcal{ALCQI}$-T$_C$Box. However, also the extended $\mathcal{ALCQI}$-T$_C$Box can still be translated into C^2 (viz. the two-variable FOL fragment with counting quantifiers [4]), by following the translation rules listed in [25, Fig.

2], plus the following rules for cardinality restrictions involving the new constructs:

$$
\begin{array}{rcl}
\Psi_x(\geqslant n\,(R \sqcup S)\,C) & := & \exists^{\geq n} y.((Rxy \vee Sxy) \wedge \Psi_y(C)) \\
\Psi_x(\geqslant n\,(R \sqcap S)\,C) & := & \exists^{\geq n} y.(Rxy \wedge Sxy \wedge \Psi_y(C)) \\
\Psi_x(\geqslant n\,(\neg R)\,C) & := & \exists^{\geq n} y.(\neg Rxy \wedge \Psi_y(C)) \\
\Psi_x(\geqslant n\,(R \Rightarrow S)\,C) & := & \exists^{\geq n} y.((\neg Rxy \vee Sxy) \wedge \Psi_y(C)) \\
\Psi_x(\geqslant n\,(R \setminus S)\,C) & := & \exists^{\geq n} y.(Rxy \wedge \neg Sxy \wedge \Psi_y(C))
\end{array}
$$

Obviously the result of the translation (obtained in linear time) will still be a sentence in C^2 and, thus, its satisfiability will still be decidable in NExpTime [17, 22]. Moreover, the addition of the new role constructors makes our Description Logics more expressive than the basic $\mathcal{ALCQIO}$ and, thus, it cannot be less complex; since it shown in [25] that $\mathcal{ALCQIO}$ reasoning is NExpTime-complete, also our $\mathcal{ALCQIOB}$ extension has the same complexity (the translation into C^2 yields an optimal solution).