Appeared in Proc. of **DOOD'95**

Structured Objects: Modeling and Reasoning

Diego Calvanese, Giuseppe De Giacomo, Maurizio Lenzerini

Dipartimento di Informatica e Sistemistica
Università di Roma "La Sapienza"
Via Salaria 113, I-00198 Roma, Italy
{calvanese,degiacomo,lenzerini}@dis.uniroma1.it

Abstract. One distinctive characteristic of object-oriented data models over traditional database systems is that they provide more expressive power in schema definition. Nevertheless, the defining power of object-oriented models is still somewhat limited, mainly because it is commonly accepted that part of the semantics of the application can be represented within methods. The research work reported in this paper explores the possibility of enhancing the power of object-oriented data models in schema definition, thus offering more possibilities to reason about the intension of the database and better supporting data management. We demonstrate our approach by presenting a new data model, called $\mathcal{CVL}$, that extends the usual object-oriented data models with several aspects, including view definition, recursive structure modeling, navigation of the schema through forward and backward traversal of links (attributes and relations), subsetting of attributes, and cardinality ratio constraints on links. $\mathcal{CVL}$ is equipped with sound, complete, and terminating inference procedures, that allow various forms of reasoning to be carried out on the intensional level of the database.

1 Introduction

One distinctive characteristic of object-oriented data models over traditional database systems is that they provide more expressive power in schema definition. Indeed, several modeling constructs of object-oriented data models are borrowed from the research on semantic data modeling and semantic networks in Artificial Intelligence, and are intended to overcome well-known limitations of flat data representation. Nevertheless, the defining power of object-oriented models is still somewhat limited. Examples of useful representation mechanisms that are considered important especially for new applications, but are generally not considered in object-oriented schemas are: recursive class definitions, view definitions, cardinality ratio constraints on attributes, subsetting of attributes, inverse of attributes, union and complement of classes (see for example [9]). One reason for limiting the expressivity of schemas is that object-oriented models support method definitions, and it is generally accepted that some of the semantics of the application could be very well represented within methods.

The research work reported in this paper explores the possibility of enhancing the power of object-oriented data models in schema definition. We argue that such enhancement is interesting from different points of view:

- Capturing more semantics at the schema level allows the designer to declaratively represent relevant knowledge about the classes of the application. It follows that sophisticated types of constraints can be asserted in the schema, rather than embedding them in methods, with the advantage of devising general integrity checking methods to be included in future database systems.
- Expressing more knowledge at the schema level implies more possibilities to reason about the intension of the database. Such reasoning can be exploited for deriving useful information for the design of the database, for the use of the database (for example in type checking), for querying purposes (e.g., in query optimization [4, 5]), and for the solution of new problems posed by cooperative and distributed information systems (for example, schema comparison and integration [8]).

In this paper, we present a new data model, called $\mathcal{CVL}$ (for Class, View, and Link), specifically designed following the above guidelines. $\mathcal{CVL}$ extends the usual expressive power of object-oriented data models by allowing:

- To specify both necessary and sufficient conditions for an object to belong to a class; necessary conditions are generally used when defining the classes that constitute the schema, whereas sufficient conditions help in the specification of views [1]. With this feature, views are part of the schema, and can be reasoned upon exactly like any other class. Note that this approach is different from considering views just as predefined queries.
- To specify complex relations that exist between classes, such as disjointness of their instances or the fact that one class equals the union of other classes;
- To refer to navigations of the schema while defining classes and views; in particular, both forward and backward navigations along relations and attributes are allowed, with the additional possibility of imposing complex conditions on the objects encountered in the navigations. Note that general navigation of the schema is possible only if the definition mechanisms supported by the data model allows one to refer to the inverse of attributes.
- To specify relations that exist between the objects reached following different links; in particular, to specify that the set of objects reached through an attribute A is included in the set of objects reached through another attribute B, thus imposing that A is a subset of B.
- To use (n-ary) relations and to declare keys on them.
- To impose cardinality ratio constraints both for attributes and for the participation of objects in relations.
- To model complex, recursive structures, and simultaneously impose several kinds of constraints on them. This feature allows the designer to define inductive structures such as lists, sequences, trees, DAGs, etc.. Although there are data models where some of these structures can be used in schema definition, $\mathcal{CVL}$ takes a much more radical approach, in that it provides the designer with a mechanism for defining his/her own structures, rather than simply adding ad hoc types.

One of the most important aspect of $\mathcal{CVL}$ is that it supports several forms of reasoning at the schema level. Indeed, the question of enhancing the expressive power of object-oriented schemas is not addressed in $\mathcal{CVL}$ by simply adding constructs to a basic object-oriented model, but by equipping the model with reasoning procedures that can make inference on the new constructs. In this sense $\mathcal{CVL}$ can be regarded as a *deductive* modeling language, but the kind of reasoning that it supports is fundamentally different from the one usually supported by deductive databases: $\mathcal{CVL}$ allows for intensional reasoning, i.e. reasoning about the schema, whereas deductive databases provide means for expressing queries in the form of logical rules and use deduction in the process of query answering.

The paper is organized as follows. In Section 2, we provide syntax and semantics of $\mathcal{CVL}$. In Section 3, we discuss the inference procedure associated with $\mathcal{CVL}$, and illustrate its use in schema level reasoning. In Section 4, we deal with the expressivity of $\mathcal{CVL}$, by showing several examples of its modeling capabilities. Finally, in Section 5, we compare $\mathcal{CVL}$ with some well-known data models, and show that it captures several important features mentioned in recent documents on the standards for object-oriented models.

2 The $\mathcal{CVL}$ data model

In this section we formally define the object-oriented model $\mathcal{CVL}$, by specifying its syntax and its semantics.

2.1 Syntax

A $\mathcal{CVL}$ *schema* is a collection of class and view definitions over an alphabet $\mathcal{B}$, where $\mathcal{B}$ is partitioned into a set $\mathcal{C}$ of *class* symbols, a set $\mathcal{A}$ of *attribute* symbols (used in record structures), a set $\mathcal{U}$ of *role* symbols (denoting binary relations over classes), and a set $\mathcal{M}$ of *method* symbols. We assume that $\mathcal{C}$ contains the distinguished elements `Any` and `Empty`[1]. In the following C, A, U and M range over elements of $\mathcal{C}$, $\mathcal{A}$, $\mathcal{U}$ and $\mathcal{M}$ respectively.

As we mentioned before, for defining classes and views we refer to complex links which are built starting from attributes and roles. An *atomic link*, for which we use the symbol l, is either an attribute, a role, or the special symbol $\ni$ (used in the context of set structures). A *basic link* b is constructed according to the following syntax rule, starting from atomic links:

$$b ::= l \mid b_1 \cup b_2 \mid b_1 \cap b_2 \mid b_1 \setminus b_2.$$

Two objects are connected by $b_1 \cup b_2$ if they are linked by b_1 or b_2, whereas two objects are connected by $b_1 \cap b_2$ ($b_1 \setminus b_2$) if they are linked by b_1 and (but not) by b_2. Finally, a *complex link* L is obtained from basic links according to:

$$L ::= b \mid L_1 \cup L_2 \mid L_1 \circ L_2 \mid L^* \mid L^- \mid \underline{\text{identity}}(C).$$

[1] We may also assume that $\mathcal{C}$ contains some additional symbols such as `Integer`, `String`, etc., that are interpreted as usual, with the constraint that no definition of such symbols appears in the schema.

Here, $L_1 \circ L_2$ means the concatenation of link L_1 with link L_2, L^* the concatenation of link L an arbitrary finite number of times, and L^- corresponds to link L taken in reverse direction. The use of $\underline{\text{identity}}(C)$ is to verify if along a certain path we have reached an object that is an instance of class C.

The distinction between basic links and complex links, is due to our attention in achieving expressivity without loosing decidability of reasoning. The unrestricted use in $\mathcal{CVL}$ of either difference or intersection on complex links would make the formalism undecidable. This can be easily proved by exploiting known undecidability results for logics of programs [17] together with the correspondence between this logics and a restricted version of $\mathcal{CVL}$ (see Section 3).

Usually, in object-oriented models every class has an associated type which specifies the structure of the value associated to an instance of the class. In $\mathcal{CVL}$, objects are not required to be of only one type. Instead, we allow for polymorphic entities, which can be viewed as having different structures corresponding to the different roles they can play in the modeled reality. Therefore we admit rather rich expressions for defining structural properties. A *structure expression*, denoted with the symbol T, is constructed as follows, starting from class symbols:

$$T ::= C \mid \neg T \mid T_1 \wedge T_2 \mid T_1 \vee T_2 \mid [A_1 \colon T_1, \ldots, A_n \colon T_n] \mid \{T\}.$$

The structure $[A_1 \colon T_1, \ldots, A_n \colon T_n]$ represents all tuples which have at least components $A_1, \ldots, A_n$ having structure $T_1, \ldots, T_n$, respectively, while $\{T\}$ represents sets of elements having structure T. Additionally, by means of $\wedge$, $\vee$, and $\neg$, we are allowed not only to include intersection and union in structure expressions (as in [2]), but also to refer to all entities that do not have a certain structure. Note that often object-oriented models make either explicitly or implicitly the assumption that every object belongs to exactly one most specific class. Under this assumption, intersection can be eliminated from the schema definition since if an object is an instance of two classes, the schema contains also a class that specializes both and of which the object is an instance of [2]. In contrast, in $\mathcal{CVL}$ we do not want to enforce the "most specific class assumption", consistently with most knowledge representation formalisms [4] and semantic data models [19]. Such assumption would also be against the spirit of our notion of polymorphism, which allows an object to simultaneously have more than one structure (and thus to belong to different unrelated classes).

Class and view definitions are built out of structure expressions by asserting constraints on the allowed links and by specifying the methods that can be invoked on the instances of the class. A class definition expresses necessary conditions for an entity to be an instance of the defined class, whereas a view definition characterizes exactly (through necessary and sufficient conditions) the entities belonging to the defined view. Our concept of view bears similarity to the concept of *query class* of [22].

Class and *view definitions* have the following forms (C is the name of the class or of the view to be defined):

<u>class</u> C	<u>view</u> C
structure-declaration	*structure-declaration*
link-declarations	*link-declarations*
method-declarations	*method-declarations*
<u>endclass</u>	<u>endview</u>

We now explain the different parts of a class (view) definition.

– A *structure-declaration* has the form

<u>is a kind of</u> T

and can actually be regarded as both a type declaration in the usual sense, and an extended ISA declaration introducing (possibly multiple) inheritance.

– *link-declarations* stands for a possibly empty set of *link-declarations*, which can further be distinguished as follows:

• *Universal-* and *existential-link-declarations* have the form

<u>all</u> L <u>in</u> T and <u>exists</u> L <u>in</u> T.

The first declaration states that each entity reached through link L from an instance of C has structure T and the second one states that for each instance of C there is at least one entity of structure T reachable through link L. Therefore such link-declarations represent a generalization of existence and typing declarations for attributes (and roles).

• A *well-foundedness-declaration* has the form:

<u>well founded</u> L.

It states that by repeatedly following link L starting from any instance of C, after a finite number of steps one always reaches an entity from which L cannot be followed anymore. Such a condition allows for example to avoid such pathological cases as a set that has itself as a member. This aspect will be discussed in more detail in section 4.

• A *cardinality-declaration* has the form:

<u>exists</u> (u, v) b <u>in</u> T or <u>exists</u> (u, v) b^- <u>in</u> T,

where u is a nonnegative integer and v is a nonnegative integer or the special value ∞. Such a declaration states for each instance of C the existence of at least u and most v different entities of structure T reachable through the basic link b (b^-)[2]. Existence and functional dependencies can be seen as special cases of this type of constraint.

• A *meeting-declaration* has the form:

<u>each</u> b_1 <u>is</u> b_2 or <u>each</u> b_1^- <u>is</u> b_2^-.

It states that each entity reachable through a link b_1 (b_1^-) from an instance o of C is also reachable from o through a different link b_2 (b_2^-). Such a declaration allows for representing inclusions between attributes, and is a restricted form of role-value map, a type of constraint commonly used in

[2] Note that requiring the link to be basic (and not generic) is essential for preserving the decidability of inference on the schema.

knowledge representation formalisms [26].[3]

- A *key-declaration* has the form:

$$\underline{\text{key}}\ A_1, \ldots, A_m, A_1'^{-}, \ldots, A_{m'}'^{-}, U_1, \ldots, U_n, U_1'^{-}, \ldots, U_{n'}'^{-}.$$

It is allowed only in class definitions and states that each entity o in C is linked to at least one other entity through each link that appears in the declaration, and moreover the entities reached through these links uniquely determine o, in the sense that C contains no other entity o' linked to exactly the same entities as o (for all links in the declaration).

– *method-declarations* stands for a possibly empty set of *method-declarations*, each having the form:

$$\underline{\text{method}}\ M\ (C_1, \ldots, C_m)\ \underline{\text{returns}}\ (C_1', \ldots, C_n').$$

It states that for each instance of C, method M can be invoked, where the type of the input parameters (besides the invoking object) that are passed to, output parameters that are returned from the method are as specified in the declaration.

2.2 Semantics

We specify the formal semantics of a $\mathcal{CVL}$ schema through the notion of *interpretation* $\mathcal{I} = (\mathcal{O}^{\mathcal{I}}, \cdot^{\mathcal{I}})$, where $\mathcal{O}^{\mathcal{I}}$ is a nonempty set constituting the *universe* of the interpretation and $\cdot^{\mathcal{I}}$ is the *interpretation function* over the universe. Note that an interpretation corresponds to the usual notion of database state. Traditional object-oriented models distinguish between objects (characterized through their object identifier) and values associated to objects. The structure of an object is specified through its value which can be either a tuple, a set or an atomic value. Since an object has a unique value it is forced to have a unique structure. Instead, in $\mathcal{CVL}$ we have chosen not to distinguish between objects and values, and we permit assigning different structures to an element of the universe of interpretation. Indeed, we regard $\mathcal{O}^{\mathcal{I}}$ as a set of *polymorphic entities*, that is entities having simultaneously possibly more than one structure, i.e.:

1. The structure of *individual*: an entity can always be considered as having this structure, and this allows it to be referenced by other objects of the domain.
2. The structure of *tuple*: an entity o having this structure can be considered as a property aggregation, which is formally defined as a partial function from $\mathcal{A}$ to $\mathcal{O}^{\mathcal{I}}$ with the proviso that o is uniquely determined by the set of attributes on which it is defined and by their values. In the sequel the term tuple is used to denote an element of $\mathcal{O}^{\mathcal{I}}$ that has the structure of tuple, and we write $[A_1\colon o_1, \ldots, A_n\colon o_n]$ to denote any tuple t such that, for each $i \in \{1, \ldots, n\}$, $t(A_i)$ is defined and equal to o_i (which is called the A_i-component of t). Note that the tuple t may have other components as well, besides the A_i-components.

[3] Note that the restricted form of role-value map adopted here does not lead to undecidability of inference, which results if this construct is used in its most general form.

3. The structure of *set*: an entity o having this structure can be considered as an instance aggregation, which is formally defined as a finite collection of entities in $\mathcal{O}^{\mathcal{I}}$, with the following provisos: (i) the view of o as a set is unique (except for the empty set $\{\}$), in the sense that there is at most one finite collection of entities of which o can be considered an aggregation, and (ii) no other entity o' is the aggregation of the same collection. In the sequel the term set is used to denote an element of $\mathcal{O}^{\mathcal{I}}$ that has the structure of set, and we write $\{\!|o_1, \ldots, o_n|\!\}$ to denote the collection whose members are exactly $o_1, \ldots, o_n$.

The interpretation function $\cdot^{\mathcal{I}}$ is defined over classes, structure expressions and links, and assigns them an *extension* as follows:

- It assigns to $\ni$ a subset of $\mathcal{O}^{\mathcal{I}} \times \mathcal{O}^{\mathcal{I}}$ such that for each $\{\!|\ldots, o, \ldots|\!\} \in \mathcal{O}^{\mathcal{I}}$, we have that $(\{\!|\ldots, o, \ldots|\!\}, o) \in \ni^{\mathcal{I}}$.
- It assigns to every role U a subset of $\mathcal{O}^{\mathcal{I}} \times \mathcal{O}^{\mathcal{I}}$.
- It assigns to every attribute A a subset of $\mathcal{O}^{\mathcal{I}} \times \mathcal{O}^{\mathcal{I}}$ such that, for each tuple $[\ldots, A\colon o, \ldots] \in \mathcal{O}^{\mathcal{I}}$, $([\ldots, A\colon o, \ldots], o) \in A^{\mathcal{I}}$, and there is no $o' \in \mathcal{O}^{\mathcal{I}}$ different from o such that $([\ldots, A\colon o, \ldots], o') \in A^{\mathcal{I}}$. Note that this implies that every attribute in a tuple is functional for the tuple.
- It assigns to every basic and complex link a subset of $\mathcal{O}^{\mathcal{I}} \times \mathcal{O}^{\mathcal{I}}$ such that the following conditions are satisfied (in the semantics, "$\setminus$" denotes set difference, "$\circ$" concatenation of binary relations, and "$*$" their reflexive transitive closure):

$$\begin{array}{rlrl} (b_1 \cup b_2)^{\mathcal{I}} &= b_1^{\mathcal{I}} \cup b_2^{\mathcal{I}} & (L_1 \circ L_2)^{\mathcal{I}} &= L_1^{\mathcal{I}} \circ L_2^{\mathcal{I}} \\ (b_1 \cap b_2)^{\mathcal{I}} &= b_1^{\mathcal{I}} \cap b_2^{\mathcal{I}} & (L^*)^{\mathcal{I}} &= (L^{\mathcal{I}})^* \\ (b_1 \setminus b_2)^{\mathcal{I}} &= b_1^{\mathcal{I}} \setminus b_2^{\mathcal{I}} & (L^-)^{\mathcal{I}} &= \{(o, o') \mid (o', o) \in L^{\mathcal{I}}\} \\ (L_1 \cup L_2)^{\mathcal{I}} &= L_1^{\mathcal{I}} \cup L_2^{\mathcal{I}} & (\underline{\text{identity}}(C))^{\mathcal{I}} &= \{(o, o) \in \mathcal{O}^{\mathcal{I}} \times \mathcal{O}^{\mathcal{I}} \mid o \in C^{\mathcal{I}}\}. \end{array}$$

- It assigns to every class and to every structure expression a subset of $\mathcal{O}^{\mathcal{I}}$ such that the following conditions are satisfied:

$$\begin{array}{rlrl} \texttt{Any}^{\mathcal{I}} &= \mathcal{O}^{\mathcal{I}} & (\neg T)^{\mathcal{I}} &= \mathcal{O}^{\mathcal{I}} \setminus T^{\mathcal{I}} \\ \texttt{Empty}^{\mathcal{I}} &= \emptyset & (T_1 \wedge T_2)^{\mathcal{I}} &= T_1^{\mathcal{I}} \cap T_2^{\mathcal{I}} \\ C^{\mathcal{I}} &\subseteq \mathcal{O}^{\mathcal{I}} & (T_1 \vee T_2)^{\mathcal{I}} &= T_1^{\mathcal{I}} \cup T_2^{\mathcal{I}} \end{array}$$

$$[A_1\colon T_1, \ldots, A_n\colon T_n]^{\mathcal{I}} = \{[A_1\colon o_1, \ldots, A_n\colon o_n] \in \mathcal{O}^{\mathcal{I}} \mid o_1 \in T_1^{\mathcal{I}}, \ldots, o_n \in T_n^{\mathcal{I}}\}$$
$$\{T\}^{\mathcal{I}} = \{\{\!|o_1, \ldots, o_n|\!\} \in \mathcal{O}^{\mathcal{I}} \mid o_1, \ldots, o_n \in T^{\mathcal{I}}\}.$$

The elements of $C^{\mathcal{I}}$ are called *instances* of C.

In order to characterize which interpretations are legal according to a specified schema we first define what it means if in an interpretation $\mathcal{I}$ an entity $o \in \mathcal{O}^{\mathcal{I}}$ *satisfies* a declaration which is part of a class or view definition:

- o satisfies a type-declaration "<u>is a kind of</u> T" if $o \in T^{\mathcal{I}}$;
- o satisfies a universal-link-declaration "<u>all</u> L <u>in</u> T" if for all $o' \in \mathcal{O}^{\mathcal{I}}$, $(o, o') \in L^{\mathcal{I}}$ implies $o' \in T^{\mathcal{I}}$;
- o satisfies an existential-link-declaration "<u>exists</u> L <u>in</u> T" if there is $o' \in \mathcal{O}^{\mathcal{I}}$ such that $(o, o') \in L^{\mathcal{I}}$ and $o' \in T^{\mathcal{I}}$;

- o satisfies a well-foundedness-declaration "well founded L" if there is no infinite chain $(o_1, o_2, \ldots)$ of entities $o_1, o_2, \ldots \in \mathcal{O}^{\mathcal{I}}$ such that $o = o_1$ and $(o_i, o_{i+1}) \in L^{\mathcal{I}}$, for $i \in \{1, 2, \ldots\}$.
- o satisfies a cardinality-declaration "exists (u, v) b in T" if there are at least u and at most v entities $o' \in \mathcal{O}^{\mathcal{I}}$ such that $(o, o') \in b^{\mathcal{I}}$ and $o' \in T^{\mathcal{I}}$; a similar definition holds for a cardinality-declaration involving b^-;
- o satisfies a meeting-declaration "each b_1 is b_2" if $\{o' \mid (o, o') \in b_1^{\mathcal{I}}\} \subseteq \{o' \mid (o, o') \in b_2^{\mathcal{I}}\}$; a similar definition holds for a meeting-declaration involving b_1^- and b_2^-.

Finally, a class C satisfies a key-declaration "key $L_1, \ldots, L_m$", if for every instance o of C in $\mathcal{I}$ there are entities $o_1, \ldots, o_m \in \mathcal{O}^{\mathcal{I}}$ such that $(o, o_i) \in L_i^{\mathcal{I}}$, for $i \in \{1, \ldots, m\}$, and there is no other entity $o' \neq o$ in $C^{\mathcal{I}}$ for which these conditions hold.

Note that the method-declarations do not participate in the set-theoretic semantics of classes and views. For an example on the use of method declarations in the definition of a schema we refer to Section 4.

An interpretation $\mathcal{I}$ *satisfies a class definition* δ, say for class C, if every instance of C in $\mathcal{I}$ satisfies all declarations in δ, and if C satisfies all key-declarations in δ. $\mathcal{I}$ *satisfies a view definition* δ, say for view C, if the set of entities that satisfy all declarations in δ is exactly the set of instances of C. In other words, there are no other entities in $\mathcal{O}^{\mathcal{I}}$ besides those in $C^{\mathcal{I}}$ that satisfy all declarations in δ.

If $\mathcal{I}$ satisfies all class and view definitions in a schema $\mathcal{S}$ it is called a *model* of $\mathcal{S}$. A schema is said to be *consistent* if it admits a model. A class (view) C is said to be *consistent in* $\mathcal{S}$, if there is a model $\mathcal{I}$ of $\mathcal{S}$ such that $C^{\mathcal{I}}$ is nonempty. The notion of consistency is then extended in a natural way to structure expressions.

3 Reasoning in $\mathcal{CVL}$

One of the main features of $\mathcal{CVL}$ is that it supports several forms of reasoning at the schema level. The basic reasoning task we consider is *consistency checking*: given a schema $\mathcal{S}$ and a structure expression T, verify if T is consistent in $\mathcal{S}$. This reasoning task is indeed the basis for the typical kinds of schema level deductions supported by object-oriented systems. In particular:

- Schema consistency: checking the consistency of a schema $\mathcal{S}$ amounts to verify if `Any` is consistent in $\mathcal{S}$.
- Class specialization: checking whether a class C is a specialization of a class C' in a schema $\mathcal{S}$ amounts to verify if $C \wedge \neg C'$ is not consistent in $\mathcal{S}$.
- Computing the class lattice of the schema, or more generally the lattice of all structure expressions: this can be performed once for all by verifying specialization between all pairs of classes (structure expressions) in the schema. Observe that such lattice can be maintained in an incremental manner.

All these inferences can be profitably exploited in both schema design and analysis (e.g. in schema integration). In a more general setting, where suitable constructs (e.g. programming language constructs) are coupled to the data model

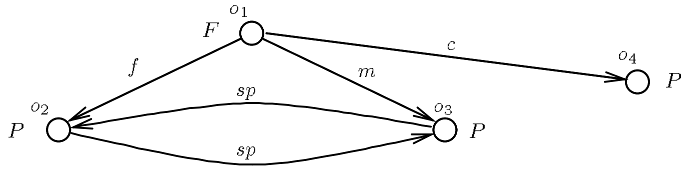

Fig. 1. Instantiation of a schema/Labeled transition system

for expressing queries and manipulation operations, these reasoning tasks provide the basis for type checking and type inference. It is outside the scope of this paper to discuss these aspects in detail, but we present an example in Section 4.

In general, schema level reasoning in object-oriented data models can be performed by means of relatively simple algorithms (see for example [21]). The richness of $\mathcal{CVL}$ makes reasoning much more difficult with respect to usual data models. Indeed the question arises if consistency checking in $\mathcal{CVL}$ is decidable at all. One of our main results is a sound, complete, and terminating reasoning procedure to perform consistency checking. The reasoning procedure works in worst-case deterministic exponential time in the size of the schema. Notably, we have shown that such worst-case complexity is inherent to the problem, proving that consistency checking in $\mathcal{CVL}$ is EXPTIME-complete.

Space limitations prevent us from exposing the details of our inference method. Here we would like to discuss the main idea, which is based on previous work relating formalisms used in knowledge representation and databases to modal logics developed for modeling properties of programs [6, 7, 12, 13].

The key point of our method is to take advantage of the strong similarity that exists between the interpretative structures of object-oriented models and labeled transition systems used in computer science to describe the behavior of program schemes. To gain some intuition on this, consider Figure 1, showing an instantiation of an object-oriented schema, where nodes correspond to objects labeled by the classes they belong to, and arcs correspond to links. Now, such instantiation can also be seen as a transition system where nodes correspond to states labeled with the properties of the state, and arcs correspond to state transitions. For example, o_1 can be seen as a state where the property F holds, and such that the execution of program f from it results in the state o_2, where P holds and F does not. Notice that the cycle involving o_2 and o_3 corresponds to a nonterminating computation.

The similarity between the interpretative structures in object-oriented models and labeled transition systems reflects in a similarity between object-oriented models and modal logics of programs, which are formalisms specifically designed for reasoning about program schemes, and which are interpreted in terms of labeled transition systems (see [20, 23] for surveys).

Such a similarity allows us to exploit the sophisticated tools available for reasoning on logics of programs, in deriving reasoning procedures for $\mathcal{CVL}$. However, the high expressivity of $\mathcal{CVL}$, and in particular the combination of cardinality

```
class Condominium
   is a kind of
      {Apartment}∧
      [loc: Address, budget: Integer]
   key loc
   exists (1, 1) manages⁻ in Manager
endclass

view CondominiumManager
   is a kind of Manager
   exists manages in Condominium
endview

class Address
   is a kind of
      [city: String, street: String,
       num: Integer]
   key city, street, num
endclass

class Manager
   is a kind of
      [ssn: String, loc: Address]
   key ssn
   exists manages in Any
endclass
```

Fig. 2. Schema of a condominium

declarations, meeting declarations and the possibility to force structures to be well-founded requires to extend the known reasoning techniques in several directions, which we now briefly sketch. Exploiting techniques developed in [11] we reduce reasoning on a schema to satisfiability of a formula of an extension of Converse-PDL, which is a well known modal logic of programs studied in [16]. The extension in obtained from Converse-PDL by including the repeat construct [24] and local functionality on direct and converse programs [12]. It is known that Converse-PDL is EXPTIME-complete, and that adding just one of the two constructs above does not increase the complexity [15, 12]. However, decidability was not known for the logic including both constructs. By extending the automata-theoretic techniques developed in [25] we have proved that such logic is decidable and EXPTIME-complete.

4 Expressivity of $\mathcal{CVL}$

In this section we discuss by means of examples the main distinguished features of $\mathcal{CVL}$ with the goal of illustrating its expressivity.

4.1 Object polymorphism

In $\mathcal{CVL}$, entities can be seen as having different structures simultaneously. In this way we make a step further with respect to traditional object models, where the usual distinction between objects (without structure) and their unique value may constitute a limitation in modeling complex application domains. As an example, in the schema of Figure 2, the structure of the class `Condominium` is specified through a conjunction of the set structure {`Apartment`} and the record structure [`loc`: `Address`, `budget`: `Integer`]. Therefore, the designer is anticipating that each instance of `Condominium` will be used both as a set (in this case the

```
view List                                class ListOfPersons
    is a kind of                             is a kind of List
        Nil ∨                                all rest* ∘ first in Person
        [first: Any, rest: List]         endclass
    exists (0, 1) rest^- in Any
    well founded rest                    class ListOfTwoPersons
endview                                      is a kind of ListOfPersons
                                             exists rest ∘ rest in Any
class Nil                                    all rest ∘ rest ∘ rest in Empty
    is a kind of Any                     endclass
    all first ∪ rest in Empty
endclass
```

Fig. 3. Schema defining lists

set of apartments forming the condominium) and as a record structure collecting the relevant attributes of the condominium (in this case where the condominium is located and its budget). Moreover, each instance of condominium can also be regarded as an individual that can be referred to by other objects through roles (in this case `manages`).

4.2 Well founded structures

In $\mathcal{CVL}$, the designer can define a large variety of recursive structures, such as lists, binary trees, trees, DAGs, streams, arrays, depending on the application need. For example, the schema in Figure 3 shows the definitions of several variants of lists. Typically, the class of lists is defined inductively as the smallest set *List* such that:

- *Nil* is a *List*, and
- every pair whose first element is any object, and whose second element is a *List*, is a *List*.

This inductive definition is captured in our model by the view `List`. This view is defined recursively, in the sense that the term `List` we are defining occurs in the body of the definition. In general, a recursive definition should not be confused with an inductive one: an inductive definition selects the smallest set satisfying a certain condition, while a recursive one simply states the condition without specifying any selection criteria to choose among all possible sets satisfying it. In fact, the well-foundedness-declaration accomplishes this selection, making our recursive definition of `List` inductive. Observe also the use of the cardinality declaration which forbids that two lists share a common sublist.

Once lists are defined in our model they can be easily specialized selecting for example the kind of information contained in an element (e.g. `ListOfPersons`) or additional structural constraints, as a specific length (e.g. `ListOfTwoPersons`).

<u>view</u> `NestedList`
  <u>is a kind of</u>
    `Nil` $\vee$
    [`first`: `Atom` $\vee$ `NestedList`,
     `rest`: `NestedList`]
  <u>exists</u> $(0,1)$ `rest`$^-$ <u>in</u> `Any`
  <u>well founded</u> `first` $\cup$ `rest`
<u>endview</u>
	<u>class</u> `Atom`
	  <u>is a kind of</u> $\neg$`Nil`
	  <u>all</u> `first` $\cup$ `rest` <u>in</u> `Empty`
	<u>endclass</u>

Fig. 4. Schema defining nested lists

Notably, recursively defined classes are taken into account like any other class definition when reasoning about the schema. Suppose for example that we define $NestedList$ as the smallest set such that:

- Nil is a $NestedList$, and
- every pair whose first element is either an $Atom$ or a $NestedList$, and whose second element is a $NestedList$, is a $NestedList$.

Such structure is captured by the definitions in Figure 4. The reasoning method correctly infers that `Atom` and `List` are disjoint and that `NestedList` is a specialization of `List`.

We argue that the ability to define recursive structures in our model is an important enhancement with respect to traditional object-oriented models, where such structures, if present at all, are ad hoc additions requiring a special treatment by the reasoning procedures [9, 3].

Well-foundedness-declarations also allow us to represent well-founded binary relations. An interesting example is the definition of the *part-of* relation, which has a special importance in modeling certain applications [10]. This relation is characterized by being finite, antisymmetric, irreflexive, and transitive. The first three properties are captured by imposing well-foundedness, while transitivity is handled by a careful use of the $*$ operator. More precisely, in order to model the part-of relation in $\mathcal{CVL}$, we can introduce a `basic_part_of` role, assert its well-foundedness for the class `Any`, and then use the link `basic_part_of` $\circ$ `basic_part_of`* as part-of. By the virtue of meeting-declarations, we can also distinguish between different specializations of the part-of relation.

4.3 Classification

We show an example of computation of the class lattice in which the reasoning procedure needs to exploit its ability to deal with recursive definitions. Figure 5 shows the definitions of classes and views concerning various kinds of directed graphs (`Graph`), including finite directed acyclic graphs (`DAG`) and finite trees (`Tree`). Our reasoning method can be used to compute the corresponding class lattice shown in Figure 6. Observe that several deductions involved in the computation of the lattice are not trivial at all. For example, in checking whether

```
class Graph
    is a kind of [label: String]
    all edge in Graph
endclass

view DAG
    is a kind of Graph
    well founded edge
endview

view Tree
    is a kind of Graph
    all edge in Tree
    well founded edge
    exists (0,1) edge⁻ in Any
endview

view BinaryGraph
    is a kind of Graph
    all edge in BinaryGraph
    exists (0,2) edge in Any
endview

view BinaryTree
    is a kind of Graph
    all edge in BinaryTree
    well founded edge
    exists (0,1) edge⁻ in Any
    exists (0,1) left in Any
    exists (0,1) right in Any
    each left ∪ right is edge
    each edge is left ∪ right
    each left is edge \ right
endview
```

Fig. 5. Schema defining graphs

`BinaryTree` is a specialization of `BinaryGraph`, a sophisticated reasoning must be carried out in order to infer that every instance of `BinaryTree` satisfies exists (0,2) `edge` in `Any`.

4.4 Methods

We already mentioned that method declarations do not participate in the set-theoretic semantics of the schema, in the sense that classification and consistency checking do not depend on them. Reasoning on methods is mostly concerned with the problem of deciding, given an object that is an instance of a certain class, and a method invocation for that object, which is the method to be called, in order to ensure that all parameters are well-typed. In making this choice, one may take advantage of the capability of reasoning on the schema.

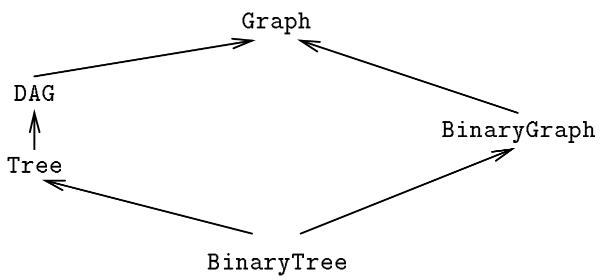

Fig. 6. A lattice of graphs

Consider, for example a schema $\mathcal{S}$ containing the following definition, where a method M is declared for class C:

$\underline{\text{class}}\ C$
  $\ldots$
  $\underline{\text{method}}\ M\ (D_1, D_2)\ \underline{\text{returns}}\ (D_3)$
  $\ldots$
$\underline{\text{endclass}}$

Suppose now that in specifying manipulations of the corresponding database we use three objects x in class C, y_1 in class D_1' and y_2 in class D_2', respectively. Let us analyze the behavior of the type checker in processing the expression

$$x.M(y_1, y_2).$$

If a strong type checking policy is enforced, then this invocation can be bound to the method defined in class C if and only if D_1' is a specialization of D_1 and D_2' is a specialization of D_2 in $\mathcal{S}$, and in this case the expression is considered well-typed. On the other hand, if a weaker type checking policy is adopted, in order to guarantee well typedness, it is sufficient that both $D_1 \wedge D_1'$ and $D_2 \wedge D_2'$ are consistent in $\mathcal{S}$. Moreover, in both cases it can be easily inferred that the type of the expression is in D_3. All these inferences can be carried out by relying on the basic reasoning task introduced in the previous section.

5 Discussion and conclusion

The combination of constructs of the $\mathcal{CVL}$ data model makes it powerful enough to capture most common object-oriented and semantic data models presented in the literature [19, 18]. In fact, by adding suitable definitions to a schema we can impose conditions that reflect the assumptions made in the various models, forcing such a schema to be interpreted exactly in the way required by each model. We show this on three relevant examples, remarking that our work focuses on modeling the structural components of a schema.

5.1 $\mathcal{CVL}$ versus O_2

We have already mentioned that object-oriented models in general, and O_2 in particular distinguish between objects characterized by their object identifier and values associated to them [3]. This dichotomy can be forced on a $\mathcal{CVL}$ schema as shown in Figure 7, where we assume that $\mathcal{C}$ contains two special elements `PureObject` and `Value`, that $\mathcal{U} = \{\texttt{hasvalue}\}$ and that $\mathcal{A} = \{A_1, \ldots, A_n\}$, where $A_1, \ldots, A_n$ are all attributes that appear in the O_2-schema we want to represent. The well-foundedness-declaration in `Value` is crucial for representing the property that record and set structures in O_2 are a priori defined to be finite.

Now, an O_2 schema $\mathcal{S}$ can be translated into a $\mathcal{CVL}$ schema by taking the definitions in Figure 7, and adding for every class C of type τ appearing in $\mathcal{S}$, the definition

```
class Any
    is a kind of PureObject ∨ Value
endclass

view PureObject
    is a kind of ¬{Any} ∧ ¬[ ]
    exists (1,1) hasvalue in Any
    all hasvalue in Value
endview

view Value
    is a kind of
        String ∨ Integer ∨ ··· ∨ {Any} ∨ [ ]
    well founded A1 ∪ ··· ∪ An ∪ ∋
endview
```

Fig. 7. Tailoring $\mathcal{CVL}$ to O_2

<u>class</u> C
  <u>is a kind of</u> `PureObject`
  <u>all</u> `hasvalue` <u>in</u> T
<u>endclass</u>,

where T is the structure expression corresponding to the O_2-type τ. Note also that disjoint object assignments (see [3]) can be imposed in $\mathcal{CVL}$ by using negation.

5.2 $\mathcal{CVL}$ versus Entity-Relationship model

The Entity-Relationship (ER) model is a semantic database model extensively used in the conceptual phase of database design [14]. The ER model distinguishes between entity-types (called simply entities in ER), denoting classes of objects, and relationships, used to model relations between entity-types. The entity-types are connected to relationships by means of ER-roles. Additionally, ER-attributes are used to associate further properties to entity-types and relationships. This setting can be represented in $\mathcal{CVL}$ as shown in Figure 8, where roles are used to represent ER-attributes and attributes to represent ER-roles.

An entity-type E_1 having two ER-attributes, and a relationship R connected through ER-roles A_1, A_2, and A_3 to entity-types E_1, E_2, and E_3, respectively, are then represented by means of:

<u>class</u> E_1
  <u>is a kind of</u> `EntityType`
  <u>all</u> U_1 <u>in</u> $Attr_1$
  <u>all</u> U_2 <u>in</u> $Attr_2$
  <u>all</u> A_1^- <u>in</u> R
  <u>exists</u> $(1,1)$ A_1^- <u>in</u> `Any`
  <u>key</u> U_1, A_1^-
<u>endclass</u>

<u>class</u> R
  <u>is a kind of</u>
    `Relationship` $\wedge [A_1 : E_1, A_2 : E_2, A_3 : E_3]$
  <u>all</u> $A_4 \cup \cdots \cup A_m$ <u>in</u> `Empty`
<u>endclass</u>

In our example E_1 has an external key constituted by U_1 and by the participation in relation R. Notice that due to the uniqueness of tuples, $\{A_1, A_2, A_3\}$ is a key for R.

class Any
  is a kind of
    $\neg$ {Any} $\wedge$
    (EntityType $\vee$ Relationship $\vee$
     Attribute)
  all $U_1 \cup \cdots \cup U_m$ in Attribute
  exists $(0,1)$ U_1 in Any
   ...
  exists $(0,1)$ U_m in Any
endclass

view EntityType
  is a kind of $\neg[\,]$
  all $A_1^- \cup \cdots \cup A_n^-$ in Relationship
endview

view Relationship
  is a kind of $[\,]$
  all $A_1 \cup \cdots \cup A_n$ in EntityType
endview

view Attribute
  is a kind of
    $\neg$EntityType $\wedge$ $\neg$Relationship
  all $A_1 \cup \cdots \cup A_m$ in Empty
  all $U_1 \cup \cdots \cup U_n$ in Empty
endview

Fig. 8. Tailoring $\mathcal{CVL}$ to the Entity-Relationship model

5.3 $\mathcal{CVL}$ versus ODMG

ODMG is intended as a standard for object-oriented models and as such it gives precise directives about the requirements a candidate object-model should possess [9]. The expressivity of $\mathcal{CVL}$ goes far beyond the one required by the current version of the standard. In fact, most of the functionality that is under consideration for the next release of the ODMG model is already captured by $\mathcal{CVL}$. This is shown by the following observations, which also serve the purpose of recalling the distinguishing features of the model we have proposed.

- In ODMG, the types are organized in a hierarchy and properties and operations for objects are inherited along this hierarchy from supertypes to subtypes. Multiple inheritance is allowed. The inheritance mechanism present in $\mathcal{CVL}$ through structure-declarations in class definitions is easily seen to accomplish the same functionality. In fact, much more complex patterns can be imposed through the unrestricted use of boolean operations in type expressions.
- ODMG distinguishes between proper objects and so called *literals*, where literals are regarded as immutable, whereas objects are created and destroyed. This distinction can be captured in our setting in a way that is similar to the one shown for handling objects and values.
- Attributes, which in ODMG relate objects to literals, and relationships, which relate objects to each other, are modeled in $\mathcal{CVL}$ through the use of roles and tuples. Referring to the traversal of relationships in both directions, which is permitted in ODMG, can be performed easily in $\mathcal{CVL}$ through the use of inverse links.
- Subtype/supertype relationships between attribute types, which are considered for future versions of ODMG, can already be modeled through meeting-

declarations.

- ODMG currently supports only binary relationships, but relationships of arbitrary arity are considered as a possible extension. $\mathcal{CVL}$ already allows to represent such relationships by means of tuples and suitable key-declarations.
- Subtype/supertype relationships between relationship types can be expressed in $\mathcal{CVL}$ through the specialization of classes whose instances are tuples.
- Structured objects such as lists and arrays, which ODMG supports as built in types, can be modeled in $\mathcal{CVL}$, as has been shown in the previous section.
- ODMG allows the definition of multiple keys, which are captured in $\mathcal{CVL}$ by key-declarations.
- In ODMG, operations supported for a certain type are specified through their signature, which defines the name of the operation and the type of its arguments and return values. This corresponds to the method-declarations in $\mathcal{CVL}$.

5.4 Concluding remarks

The comparison presented in this section shows that $\mathcal{CVL}$ indeed provides powerful representation mechanisms that can be specialized so as to capture existing approaches to object-oriented data modeling. It is worth reminding that $\mathcal{CVL}$ is equipped with reasoning procedures that can be exploited in various ways in the use of the database. In this paper, we have described the basic reasoning method for consistency checking. Future work on $\mathcal{CVL}$ will be devoted to refine such method in order to devise effective algorithms for schema analysis and design, schema integration, type checking, type inference, and query optimization, both in general, and in the specialized frameworks discussed in this section.

Acknowledgements. This work has been partially supported by the ESPRIT Basic Research Action N.6810 (COMPULOG 2).

References

1. S. Abiteboul and A. Bonner. Objects and views. In J. Clifford and R. King, editors, *Proc. of ACM SIGMOD*, pages 238–247, 1991.
2. S. Abiteboul and P. Kanellakis. Object identity as a query language primitive. In *Proc. of ACM SIGMOD*, pages 159–173, 1989.
3. F. Bancilhon, C. Delobel, and P. Kanellakis. *Building an Object-Oriented Database System – The story of O_2*. Morgan Kaufmann, 1992.
4. S. Bergamaschi and C. Sartori. On taxonomic reasoning in conceptual design. *ACM Trans. on Database Systems*, 17(3):385–422, 1992.
5. M. Buchheit, M. A. Jeusfeld, W. Nutt, and M. Staudt. Subsumption between queries to Object-Oriented databases. *Information Systems*, 19(1):33–54, 1994.
6. D. Calvanese and M. Lenzerini. Making object-oriented schemas more expressive. In *Proc. of PODS-94*, pages 243–254. ACM Press and Addison Wesley, 1994.
7. D. Calvanese, M. Lenzerini, and D. Nardi. A unified framework for class based representation formalisms. In J. Doyle, E. Sandewall, and P. Torasso, editors, *Proc. of KR-94*, pages 109–120. Morgan Kaufmann, 1994.

8. T. Catarci and M. Lenzerini. Representing and using interschema knowledge in cooperative information systems. *J. of Intelligent and Cooperative Information Systems*, 2(4):375–398, 1993.
9. R. G. G. Cattell, editor. *The Object Database Standard: ODMG-93.* Morgan Kaufmann, 1994. Release 1.1.
10. V. Christophides, S. Abiteboul, S. Cluet, and M. Scholl. From structured documents to novel query facilities. In R. T. Snodgrass and M. Winslett, editors, *Proc. of ACM SIGMOD*, pages 313–324, 1994.
11. G. De Giacomo. *Decidability of Class-Based Knowledge Representation Formalisms and their Application to Medical Terminology Servers.* PhD thesis, Dip. di Inf. e Sist., Univ. di Roma "La Sapienza", 1995.
12. G. De Giacomo and M. Lenzerini. Boosting the correspondence between description logics and propositional dynamic logics. In *Proc. of AAAI-94*, pages 205–212. AAAI Press/The MIT Press, 1994.
13. G. De Giacomo and M. Lenzerini. What's in an aggregate: Foundations for description logics with tuples and sets. In *Proc. of IJCAI-95*, 1995.
14. G. Di Battista and M. Lenzerini. Deductive entity-relationship modeling. *IEEE Trans. on Knowledge and Data Engineering*, 5(3):439–450, 1993.
15. E. A. Emerson and C. S. Jutla. On simultaneously determinizing and complementing ω-automata. In *Proc. of LICS-89*, pages 333–342, 1989.
16. M. J. Fischer and R. E. Ladner. Propositional dynamic logic of regular programs. *J. of Computer and System Sciences*, 18:194–211, 1979.
17. D. Harel. Dynamic logic. In *Handbook of Philosophical Logic*, volume 2, pages 497–640. D. Reidel, Dordrecht, Holland, 1984.
18. R. Hull. A survey of theoretical research on typed complex database objects. In J. Paredaens, editor, *Databases*, pages 193–256. Academic Press, 1988.
19. R. B. Hull and R. King. Semantic database modelling: Survey, applications and research issues. *ACM Computing Surveys*, 19(3):201–260, Sept. 1987.
20. D. Kozen and J. Tiuryn. Logics of programs. In J. V. Leeuwen, editor, *Handbook of Theoretical Computer Science – Formal Models and Semantics*, pages 789–840. Elsevier Science Publishers (North-Holland), 1990.
21. C. Lecluse and P. Richard. Modeling complex structures in object-oriented databases. In *Proc. of PODS-89*, pages 362–369, 1989.
22. M. Staudt, M. Nissen, and M. Jeusfeld. Query by class, rule and concept. *J. of Applied Intelligence*, 4(2):133–157, 1994.
23. C. Stirling. Modal and temporal logic. In S. Abramsky, D. M. Gabbay, and T. S. E. Maibaum, editors, *Handbook of Logic in Computer Science*, pages 477–563. Clarendon Press, 1992.
24. R. E. Streett. Propositional dynamic logic of looping and converse is elementarily decidable. *Information and Computation*, 54:121–141, 1982.
25. M. Y. Vardi and P. Wolper. Automata-theoretic techniques for modal logics of programs. *J. of Computer and System Sciences*, 32:183–221, 1986.
26. W. A. Woods and J. G. Schmolze. The KL-ONE family. In F. W. Lehmann, editor, *Semantic Networks in Artificial Intelligence*, pages 133–178. Pergamon Press, 1992. Published as a special issue of *Computers & Mathematics with Applications*, Volume 23, Number 2–9.

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