Increasing the Power of Structured Objects

Diego Calvanese and **Giuseppe De Giacomo** and **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

1 Introduction

We have recently proposed a new object-oriented data model, called $\mathcal{CVL}$ (for Classes, Views, and Links), that extends the expressive power of known formalisms in several directions by offering the following possibilities:

- 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 the specification of views requires to state conditions that are both necessary and sufficient [1]. With this feature, supported in $\mathcal{CVL}$ through class and view definitions, views are part of the schema and can be reasoned upon exactly like any class.
- 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.
- 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 with complex properties and to declare keys on them.
- To impose cardinality ratio constraints on attributes.
- To model complex, recursive structures, simultaneously imposing several kinds of constraints on them. This feature allows the designer to define inductive structures such as lists, sequences, trees, DAGs, etc..

One of the most important aspects of the model we propose 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 more and more constructs to a basic object-oriented model, but by equipping the model with reasoning procedures which are able to make inference on the new constructs. Notably, we have shown that the main reasoning task in $\mathcal{CVL}$, namely checking if a schema is consistent, is decidable, by providing a sound and complete algorithm that works in worst-case deterministic exponential time in the size of the schema. Such worst-case complexity is inherent to the problem, proving that consistency checking in $\mathcal{CVL}$ is EXPTIME-complete.

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, a set $\mathcal{U}$ of *role* symbols, 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 through 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 through b_1 and (but not) by b_2. Finally, a generic *complex link* L is obtained from basic links according to:

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

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.

[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.

Usually, in object-oriented models to every class there is an associated type which specifies the structure of the value associated to each instance of the class. In $\mathcal{CVL}$, objects are not required to be of only one specified 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:

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

The structure $[A_1{:}T_1,\ldots,A_n{:}T_n]$ 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, since we allow for entities having multiple structure, intersection cannot be eliminated from the definition of structure expressions (contrast this property with the model presented in [2]).

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 [14].

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

```
class C                          view C
  structure-declaration            structure-declaration
  link-declarations                link-declarations
  method-declarations              method-declarations
endclass                         endview
```

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

(i) A *structure-declaration* has the form

$$\underline{\text{is a kind of}}\ T$$

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

(ii) 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

$$\underline{\text{all}}\ L\ \underline{\text{in}}\ T \quad \text{and} \quad \underline{\text{exists}}\ L\ \underline{\text{in}}\ T.$$

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:

$$\underline{\text{well founded}}\ L.$$

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:

$$\underline{\text{exists}}\ (u,v)\ b\ \underline{\text{in}}\ T \quad \text{or} \quad \underline{\text{exists}}\ (u,v)\ b^-\ \underline{\text{in}}\ T,$$

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:

$$\underline{\text{each}}\ b_1\ \underline{\text{is}}\ b_2 \quad \text{or} \quad \underline{\text{each}}\ b_1^-\ \underline{\text{is}}\ b_2^-.$$

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 knowledge representation formalisms [15].[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).

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

[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.

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. Differently from traditional object-oriented models, we do not distinguish between objects (characterized through their object identifier) and values associated to objects. Instead, we regard $\mathcal{O}^\mathcal{I}$ as being a set of *polymorphic entities*, which means that every element of $\mathcal{O}^\mathcal{I}$ can be seen as having one or both of the following structures (entities having none of these structures are called *pure objects*):

(1) The structure of *tuple*: when an entity o has this structure, it 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 $[A_1\!: o_1, \ldots, A_n\!: o_n]$ 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.

(2) The structure of *set*: when an entity o has this structure, it 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 $[\ldots, A\!: o, \ldots] \in \mathcal{O}^\mathcal{I}$, $([\ldots, A\!: o, \ldots], o) \in A^\mathcal{I}$, and there is no $o' \in \mathcal{O}^\mathcal{I}$ different from o such that $([\ldots, A\!: o, \ldots], o') \in A^\mathcal{I}$. Note that this implies that every attribute in a tuple is functional for the tuple.
• It assigns to every link a subset of $\mathcal{O}^\mathcal{I} \times \mathcal{O}^\mathcal{I}$ such that the following conditions are satisfied:

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

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

$$\begin{array}{rcl}
\texttt{Any}^\mathcal{I} & = & \mathcal{O}^\mathcal{I} \\
\texttt{Empty}^\mathcal{I} & = & \emptyset \\
C^\mathcal{I} & \subseteq & \mathcal{O}^\mathcal{I} \\
(\neg T)^\mathcal{I} & = & \mathcal{O}^\mathcal{I} \setminus T^\mathcal{I} \\
(T_1 \wedge T_2)^\mathcal{I} & = & T_1^\mathcal{I} \cap T_2^\mathcal{I} \\
(T_1 \vee T_2)^\mathcal{I} & = & T_1^\mathcal{I} \cup T_2^\mathcal{I} \\
[A_1\!: T_1, \ldots, A_n\!: T_n]^\mathcal{I} & = & \{[A_1\!: o_1, \ldots, A_n\!: o_n] \in \mathcal{O}^\mathcal{I} \mid \\
 & & \quad o_1 \in T_1^\mathcal{I}, \ldots, o_n \in T_n^\mathcal{I}\} \\
\{T\}^\mathcal{I} & = & \{\{|o_1, \ldots, o_n|\} \in \mathcal{O}^\mathcal{I} \mid \\
 & & \quad o_1, \ldots, o_n \in T^\mathcal{I}\}.
\end{array}$$

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 "is a kind of T" if $o \in T^\mathcal{I}$;
• o satisfies a universal-link-declaration "all L in 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 "exists L in 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, such as checking schema consistency and class subsumption, and computing the class lattice of the schema. All these inferences can be profitably exploited in both schema design and analysis (for example in schema integration) and also provide the basis for type checking and type inference.

In general, schema level reasoning in object-oriented data models can be performed by means of relatively simple algorithms (see for example [13]). 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 our inference method, which is based on previous work relating formalisms used in knowledge representation and databases to modal logics developed for modeling properties of programs [5, 9, 10]. For more details we refer to [4].

4 Expressivity of $\mathcal{CVL}$

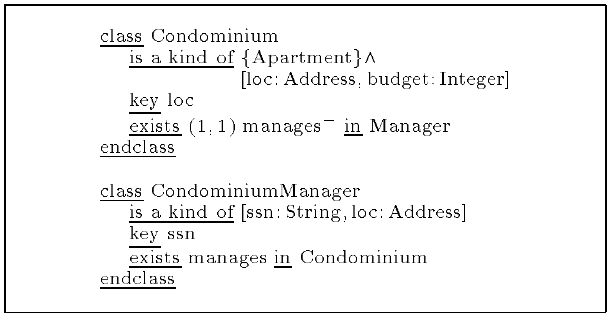
In this section we discuss by means of examples the main distinguished features of $\mathcal{CVL}$ with the goal of illustrating its expressivity.

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, `Condominium` in the schema of Figure 1 is regarded as a set of apartments, as a record structure collecting all its relevant attributes and as an object that can be referred to by other objects through roles (in our example `manages`).

4.2 Well founded structures

In $\mathcal{CVL}$, the designer can define a large variety of finite recursive structures, such as lists, binary trees,



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

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

Figure 1: Schema of a condominium

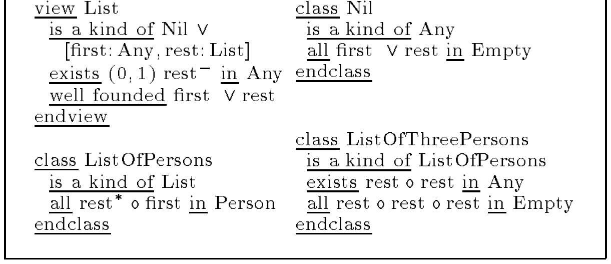
trees, directed acyclic graphs, arrays, depending on the application need. The schema in Figure 2 shows an example of definitions of several variants of lists. Observe the importance of the well-foundedness-declaration in the definition of `List`.

Notably, recursively defined classes are taken into account like any other class definition when reasoning about the schema. We argue that the ability to define finite 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 [6, 3].

Well-foundedness-declarations also allow us to represent well-founded binary relations. An interesting example of such possibility is the definition of the *part-of* relation, which has a special importance in modeling complex applications [8]. 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. Notice that 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 3 shows the definitions of classes and views concerning various kinds of (directed) graphs. Our reasoning method can be used to compute the corresponding class lattice shown in Figure 4. Observe



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

Figure 2: Schema defining lists

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

view FiniteDAG
  is a kind of Graph
  well founded edge
endview

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

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

view FiniteBinTree
  is a kind of Graph
  all edge in FiniteBinTree
  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
```

Figure 3: Schema defining graphs

that several deductions involved in the computation of the lattice are not trivial at all. For example, in computing subsumption between `FiniteBinTree` and `BinGraph`, a sophisticated reasoning must be carried out in order to infer that every instance of `FiniteBinTree` satisfies exists (0,2) `edge` in `Any`.

4.4 Methods

Consider a schema $\mathcal{S}$ in which the definition of a class C contains the method declaration "method M (D_1, D_2) returns (D_3)". 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 the type checker follows a strong type checking policy, then the expression would be considered well typed if and only if D'_1 is subsumed by D_1 and D'_2 is subsumed by D_2 in $\mathcal{S}$. 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 Concluding remarks

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 [12, 11], such as O_2 [3], ODMG [6], and the entity-relationship model [7]. 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.

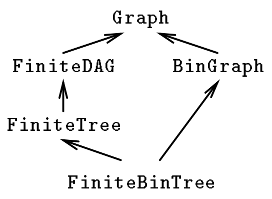


Figure 4: A lattice of graphs

References

[1] S. Abiteboul and A. Bonner. Objects and views. In J. Clifford and R. King, editors, *Proc. of ACM SIGMOD*, pages 238–247, New York (NY, USA), 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, Los Altos, 1992.

[4] D. Calvanese, G. De Giacomo, and M. Lenzerini. Structured objects: Modeling and reasoning, 1995. To appear in Proc. of DOOD-95.

[5] 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, Bonn, 1994. Morgan Kaufmann, Los Altos.

[6] R. G. G. Cattell, editor. *The Object Database Standard: ODMG-93*. Morgan Kaufmann, Los Altos, 1994. Release 1.1.

[7] P. P. Chen. The Entity-Relationship model: Toward a unified view of data. *ACM Trans. on Database Systems*, 1(1):9–36, Mar. 1976.

[8] 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, Minneapolis (Minnesota, USA), 1994.

[9] 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.

[10] 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. To appear.

[11] R. Hull. A survey of theoretical research on typed complex database objects. In J. Paredaens, editor, *Databases*, pages 193–256. Academic Press, 1988.

[12] R. B. Hull and R. King. Semantic database modelling: Survey, applications and research issues. *ACM Computing Surveys*, 19(3):201–260, Sept. 1987.

[13] C. Lecluse and P. Richard. Modeling complex structures in object-oriented databases. In *Proc. of PODS-89*, pages 362–369, 1989.

[14] M. Staudt, M. Nissen, and M. Jeusfeld. Query by class, rule and concept. *J. of Applied Intelligence*, 4(2):133–157, 1994.

[15] 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.