A Formal Model of Views for Object-Oriented Database Systems

Giovanna Guerrini*
Dipartimento di Informatica e Scienze dell'Informazione, Università degli Studi di Genova, Via Dodecaneso, 35 - 16146 Genova, Italy. E-mail: guerrini@disi.unige.it

Elisa Bertino, Barbara Catania
Dipartimento di Scienze dell'Informazione, Università degli Studi di Milano, Via Comelico 39/41 - 20135 Milano, Italy. E-mail: bertino/catania@dsi.unimi.it

Jesus Garcia-Molina†
Departamento de Informatica y Sistemas, Universidad de Murcia, Campus de Espinardo - 30071 Espinardo, Murcia, Spain. E-mail: jmolina@fcu.um.es

The definition of a view mechanism is an important issue for object-oriented database systems, in order to provide a number of features that are crucial for the development of advanced applications. Due to the complexity of the data model, the object-oriented paradigm introduces new problems in the definition of a view mechanism. Several approaches have been defined, each defining a particular view mechanism tailored to a set of functionalities that the view mechanism should support. In particular, views can be used as shorthand in queries, can support the definition of external schemas, can be used for content-dependent authorization, and, finally, can support some form of schema evolution. In this paper, we formally introduce a view model for object-oriented databases. Our view model is comparable to existing view models for what concerns the supported features; however, our model is the only one for which a formal definition is given. This formal definition of object-oriented view mechanisms is useful both for understanding what views are and as a basis for further investigations on view properties. The paper introduces the model, discussing all the supported features both from a theoretical and practical point of view. A comparison of our model with other models is also presented.

1. Introduction

The object-oriented paradigm has been recognized as a sound basis for a new generation of database systems. Indeed, its ability to model complex objects, together with its modularity and extensibility properties, overcomes most of the problems arising in the use of the simple relational model [10]. A general agreement exists on the fact that object-oriented database features should meet as much as possible functionalities of the relational database systems [5, 7, 25]. Over the last years, a considerable research effort has been devoted to explore how the paradigm shift from the relational data model to an object-oriented model affects notions such as query languages, authorization, indexing, schema evolution and concurrency control.

An important relational functionality is represented by views. In the relational model, a view is a virtual (i.e. not physically stored) relation, defined by a query on one or more stored relations. As relational languages are closed (i.e. the result of a query expressed in a relational language is a relation), the relation returned by such a query represents the view content. Thus, relational views can be used in (almost) any context in which a relation may appear. Moreover, authorizations may be granted and revoked on views as on ordinary relations. At the same time, views can be used to define external schemas, in that virtual relations are generated by combining base relations. Views are an integral component of the ANSI three-level schema architecture standard that has driven the construction and use of relational database systems. Such a schema architecture consists of the storage schema describing the storage structures for a database, the conceptual schema describing the logical model of the database, and the

*The work of Giovanna Guerrini has been partially supported by the EEC under ESPRIT Project 6333 IDEA.
†The work of Jesus Garcia-Molina has been supported by the DGICYT (Ministerio de Educacion y Ciencias, Spain) grant PR94-286.

external schema describing the derived views of the conceptual schema for particular users or group of users.

The definition of a view mechanism has been recognized to be a fundamental aspect also for the practical development of object-oriented applications. In this new context, views should be still used as shorthand in queries, should support the integration of heterogeneous databases, should be the basis for content-dependent authorization and, finally, should support the simulation of schema changes. Indeed, as it has been recognized by several researchers [8, 11, 34], views allow to dynamically modify a database schema yet retaining its older versions, a very important capability for advanced applications [24]. Moreover, a foundation for external schemas will complete the development of a three-level schema architecture for object-oriented database systems, comparable to that for relational database systems.

Unfortunately, the definition of an object-oriented view mechanism does not come for free from the relational approach. The main problems in the definition of an object-oriented view mechanism can be summarized as follows:

1. The object-oriented model is far more complex than the relational one. Whereas a relational schema consists of a set of independent relations, an object-oriented schema is a class hierarchy, where classes are connected by inheritance relationships. A view model should provide an answer to the question: *How are views integrated in the existing class hierarchy?*
2. Objects have an identity. A view, at the data level, should be a class. But what are view instances? Are they values, or existing objects, or newly generated objects? Note that this problem does not arise in the relational model, where no strong identity concept, such as the object identifier, is modeled.

Several approaches have been proposed to model views in object-oriented database systems [1, 8, 36, 39, 42] (see [30] for a survey). They address the previous issues according to different approaches. In general, besides being based upon different data models and exploiting different query languages to express view populations, the proposals differ for: the set of functionalities supported by the view mechanism (shorthand in queries, external schema definition, schema evolution, authorization, etc.); the approach with respect to the placement of views in the schema; the properties assigned to views objects (i.e., whether or not persistent object identifiers are provided); the update operations allowed on views. An optimal solution to the view mechanism does not exists. Rather, some *good* solutions can be defined for each class of chosen functionalities. Thus, a view mechanism defined to support schema evolution may be different from a view mechanism defined for only using views as shorthand in queries.

The aim of this paper is the formal definition of a view mechanism in the context of the Chimera data model [21]. Though the view mechanism has been proposed for a particular object-oriented data model, the basic concepts of our view mechanism can be applied to other data models as well such as O_2[19], GemStone [12], as well as to the ODMG standard [15]. The choice of Chimera as reference data model is mainly due to the facts that (*i*) a formal specification for Chimera exists; (*ii*) the model is at the same time deductive, active and object-oriented. This allows to investigate new insights in the context of object-oriented view mechanisms, such as the use of logical languages as a basis for defining views. Note that another interesting topic is related to the use of other Chimera capabilities, such as logical integrity constraints and triggers, in view definition. This topic is however left to further research.

Our view mechanism is comparable to (or includes) existing ones [1, 8, 36, 39, 42] for what concerns the supported features; however, our model is the only one for which a formal definition is given. Because of the similarity of features, our definition can be adapted to other object-oriented view mechanisms and, thus, has value beyond the particular view mechanism we propose. Following [8], we agree with the requirement that a view mechanism must support schema evolution. Moreover, we believe that a view mechanism should allow the definition of external schemas, as a basis for developing object-oriented applications. That requires that a view, at class level, must be usable in any context in which a class may appear. The main features of our approach thus strictly depend on those choices[1]. In particular:

- In defining a view, the user can choose among *object-preserving views*, *object-generating views* or *set-tuple views*, depending on whether the view is populated with objects extracted from an existing class, or the view must be instantiated with new objects, or the view instances do not require persistent object identifiers. Set-tuple views allow to support relations in the object data model, thus meeting the requirements of relational object models such as UniSQL [26] or Matisse [2].
- Following [8], we do not integrate views in the class inheritance hierarchy. Rather, views are organized in a separate hierarchy: they are related by a view inheritance relationship which is analogous to the inheritance relationship on classes. Moreover, the schema is extended with a new relationship, called *view derivation* relationship, connecting a view with the classes from which it is derived. The view derivation hierarchy is orthogonal to the class inheritance hierarchy.
- Two view levels are devised: views and schema views. Views are virtual classes and can be used in

any context in which classes can be used; schema views provide the capability of restructuring a schema so that it meets the need of specific applications. A schema view is a virtual schema, that is, a schema which consists of views rather than of classes.

Thus, our model basically extends the view model presented in [8] with object-preserving views, the view inheritance relationship and the concept of schema view. Moreover, we analyze in depth our approach to the placement of views in schemas, based on the view derivation relationship.

The contribution of this work is, besides defining a view model for Chimera, the development of a formal framework within which the main issues concerning the definition of a view model are systematically organized and formally described. In our opinion, a formal definition for a view model is a useful contribution. It is crucial in clearly and unambiguously specifying the features of the view mechanism and it is a foundation based on which properties about views (e.g., update propagation, view maintenance) can be formally stated and, possibly, better investigated. In particular, based on this model, we have formally defined: several notions of consistency for view instances and databases; well-formedness conditions for view inheritance hierarchies; the notion of view schema closure with respect to aggregation and inheritance hierarchies. To our knowledge, the above notions have never been formally defined.

This paper is organized as follows. Section 2 briefly describes the Chimera language, introducing the Chimera concepts relevant to our view model. In Section 3, our design choices are discussed and compared with the most relevant approaches presented in the literature. The view definition language is described in Section 4. Sections 5 and 6 present the formal specification of the view model proposed for Chimera; in particular, Section 5 introduces views while Section 6 is devoted to schema views. Finally, Section 7 presents some conclusions and outlines future work.

2. Chimera

Chimera integrates an object-oriented data model, a declarative query language based on deductive rules and an active rule language for reactive processing[2]. In what follows, we first introduce the basic notions of the data model, then present its deductive query language.

2.1. *Chimera object-oriented data model*

Chimera provides all concepts commonly ascribed to object-oriented data models. It is worth noting the following features:

- Like other object-oriented data models (e.g. O_2[19]), Chimera provides both the notions of values and types and the notions of objects and classes. Values are instances of types and are manipulated by primitive operators. Values can be primitive or complex. Each class is associated with a type describing the structure of the class instances. Moreover, in order to type variables that have to be instantiated with objects instances of a given class, class names are allowed as types.
- Object attributes can be derived, that is, defined by deductive rules.
- The implementation of methods may be specified by an update rule, that is, a rule containing a sequence of update primitives whose execution is constrained by a declarative formula, or may be external, implemented in some programming language.
- Multiple inheritance and multiple class instantiation are supported. Thus, an object can belong to several classes, even classes not related in the inheritance hierarchy.
- Classes are objects. Therefore a class definition can include class attributes, methods and constraints that collectively apply to the class.
- Each class has both intensional and extensional nature.

In the remainder of this section, we recall the aspects of the Chimera data model relevant to this work. A complete formal definition of the model can be found in [21].

The set of Chimera types $\mathcal{T}$ (that are collection of values) is defined as the union of value types ($\mathcal{VT}$) and object types ($\mathcal{OT}$). Object types are class names and their instances are object identifiers. Value types can be either basic domains (integers, reals, booleans, characters, strings) or structured types built by applying the set, list or record constructors to value or object types. Object types are class names. A Chimera class definition consists of two components: the *signature*, specifying all the information that the user must know for using the class, and the *implementation*, providing an implementation for the signature. The signature consists of a number of clauses, including the name of the superclasses and the specification of the *class features*: instance and class attributes, instance and class operations, instance and class constraints, and triggers. The signature also specifies for each attribute whether the attribute is derived or not. The implementation of a class must specify an implementation for all derived attributes, operations, constraints, at instance as well as at class level, and triggers that are specified in the signature.

A Chimera class signature is characterized by a structural and a behavioral component, specifying the signature of attributes and methods for objects instances of that class. In addition, a constraint component contains the signature of the constraints on class instances. Being class attributes supported in Chimera, a class is also characterized by a time-varying state, whose structure is specified in the corresponding metaclass. Finally, a class is characterized by an *extent* and a *proper_extent*, denoting the set of all the oids of members of the class and the oids of instances of the class, respectively. We recall that, according to the usual terminology, an object is an *instance* of a class if that class is the most specific one, in the inheritance hierarchy, to which the object belongs. Whenever an object o is an instance of a class c then o is also a *member* of all the superclasses of c.

In addition to a signature, classes have an implementation. In a Chimera class implementation, derived attributes and constraints are implemented by means of deductive rules specifying the computation of values, and the implementation of an operation is an expression of the form[3]

$$op\_name : condition \rightarrow op\_code$$

where *op_name* is the operation name applied to a list of parameters, *condition* is a Chimera formula, specifying a declarative control upon operation execution, while *op_code* is a sequence of update primitives (object creation and deletion, object migration from one class to another and state changes). Side-effect free operations can be expressed in Chimera by rules consisting only of a condition without *op_code* part. They can be useful to compute derived data.

Given a type $T \in \mathcal{T}$, its extension $[\![T]\!]$ is defined as the set of legal values for that type. The extension of types, like classes, with an explicit time-varying extent, is that extent. In particular, for an object type c, $[\![c]\!]$ is the set of oids of members of class c. Starting from the extensions of predefined basic types, which are postulated, the extensions of other value types are defined in a quite straightforward way [21].

A Chimera object is characterized by an immutable identifier and a state. The set of classes to which the object belongs as an instance is associated with each object. Each object is required to be instance of one class.

Chimera provides multiple inheritance and multiple class instantiation. Inheritance relationships among classes are described by an ISA hierarchy established by the user. This ISA hierarchy represents which classes are subclasses of (inherit from) other classes. A set of conditions must be satisfied by two classes related by the ISA relationship. These conditions are related to the fact that each subclass must contain all attributes, operations, constraints (both on the class as well on the instance level) of all its superclasses. Apart from the inherited concepts, additional features can be introduced in a subclass. Inherited concepts may be redefined (overwritten) in a subclass definition under a number of restrictions. Indeed, in Chimera the redefinition of the signature of an attribute is possible by specializing the domain of the attribute. The redefinition of the signature of an operation must verify the *covariance rule* for result parameters and the *contravariance rule* for the input ones. Therefore, result parameter domains may be specialized, whereas input parameter domains may be generalized, in the subclass signature of the operation. The implementation of an attribute or an operation may be redefined as well, introducing a different implementation of the respective concept, which "overrides" the inherited definition. The redefinition of derived and extensional attributes is not allowed if a derived attribute becomes extensional or vice-versa. Constraint redefinition is not currently allowed in Chimera. We also require that the extent of a subclass is a subset of the extent of all its superclasses.

While the redefinition of operations does not hinder the type safety of the language, the redefinition of attributes must be considered carefully [21]. The covariant redefinition of attributes (the domain of an attribute may be specialized in subclasses) reflects what is usually needed when creating a taxonomy of classes; indeed, when specializing a class the designer usually needs to add new attributes or to specialize existing ones. The problems arising when attributes are redefined in a covariant way along the inheritance hierarchy have been first recognized by Cardelli [14]. The approach adopted in Chimera is to consider the domains of attributes as integrity constraints, thus checked runtime, rather than dealing with them as type constraints, to be checked statically. Thus, whenever a value is assigned to an object attribute we dynamically check that the value is appropriate for the domain.

At the intensional level (schema level) the ordering on classes imposed by the ISA hierarchy is said to be well-defined (*int-well-defined* [21]) if each subclass contains all the features of the superclasses, possibly redefined as sketched previously. At the extensional level (instance level), the ordering on classes imposed by the ISA hierarchy is said to be well-defined (*ext-well-defined* [21]) if it is consistent with the set inclusion relationship on class extents.

A Chimera *base schema* is a set of classes, related by inheritance and aggregation relationships, modeling the structural and behavioral aspects of the problem domain. A base schema is the database initial schema defined by the system administrator, on which the object database is created. An object database is a consistent set of objects, coupled with two functions, one, referred to as *oid assignment*, handles class extents, that is, maps objects to classes, while the other one assigns values to class attributes. For an object database to be consistent, each object must belong to a class defined in the schema, each object state must contain a legal value for each attribute of each class the object belongs to, and must meet each constraint in such classes; finally, the ISA ordering is required to be ext-well-defined. Given a base schema, S, the term *base object database* will denote an object database that is instance of S; the objects in that database will be referred to as *base objects*.

2.2. Chimera formulas and rules

In this subsection we introduce Chimera rules, which are a mean to express declarative conditions on a database. Besides being used to specify the implementation of different class features, Chimera rules are used to express queries. First, we consider the set of Chimera terms, which is inductively defined as follows:

- variables are terms;
- values (basic and complex ones), excepts oids[4], are terms;
- path expressions (built making use of the dot notation) are terms; path expressions may contain attribute accesses and method invocations, provided that the invoked method is side-effect free[5].

In addition, a number of terms obtained using classical predefined operators for integers, reals, lists and sets are considered.

Chimera atomic formulas are built by applying a predicate symbol to a list of parameter terms. As stated by the following definition, we consider three kinds of atomic formulas[6].

Definition 1 (Atomic Formulas) [21]. *Chimera atomic formulas are defined as follows:*

- if t_1, t_2 are terms and $op \in \{<, >, \geq, \leq, =, ==, ==_d\}$[7] is a predefined predicate, then $t_1 op t_2$ is a comparison formula;
- if t_1, t_2 are terms, or if t_1 is a term and $t_2 \in \mathcal{CI}$ is a class name, then $t_1 in t_2$ is a membership formula;
- if t is a term and c is a class (or type) name, then $c(t)$ is a class formula. □

Complex formulas (or simply formulas) are obtained from atomic formulas and negated atomic formulas by means of conjunctions. All variables are assumed to be implicitly quantified as in Datalog [16].

Definition 2 (Formulas) [21]. *Formulas are inductively defined as follows:*

- all atomic formulas are formulas;
- if F is an atomic comparison or membership formula[8], then $\neg F$ is a (complex) formula;
- if F_1 and F_2 are formulas, then $F_1 \wedge F_2$ is a (complex) formula. □

Definition 3 (Rules) [21]. *A Chimera rule is an expression of the form*

$$Head \leftarrow Body$$

where Head is an atomic formula and Body is an arbitrary formula, such that each variable in Head occurs in Body and Body contains exactly one class formula for every variable appearing in the rule. □

The interested reader can find additional details on Chimera rules and their semantics in [21].

3. Dimensions in view design

In this section, we discuss the main dimensions in the design of a view mechanism. For each dimension, we contrast our choice with the ones made by most relevant view models in the literature.

Besides choosing a reference data model and a query language, the main design choices concern:

- how views are inserted in the database schema;
- whether views are only populated with base objects (object-preserving views), or it is possible to populate a view by creating new objects (object-generating views).

The choices to be taken with respect to those dimensions are strongly influenced by the functionalities to be supported by the view mechanism. In what follows, we first present the goals of the proposed model, and analyze their implications on the view mechanism. Next, we analyze the two main dimensions, showing how the chosen objectives affect our choices. Subsection 3.4 concludes the discussion by summarizing in Table 1 the comparison among our model and other view models proposed in the literature.

3.1. View functionalities

Whereas relational views have been used for external schema definition, data protection (content-based authorizations) and shorthand for queries, object-oriented views can be exploited also for other kinds of functionality, such as supporting schema evolution and integrating heterogeneous databases. The use of views for integrating heterogeneous database schemas has been considered in [22, 27]: a view definition integrates semantically equivalent classes belonging to different schemas. The use of views to simulate schema evolution, allowing the users to experiment with schema changes without affecting other users, was first proposed in [8]. Views can support the implementation of a schema versioning mechanism, such that any object stored in the database can be accessed and modified from any schema version including a view of the object class. Recently, other view models have considered that use of views [11, 27, 34]. The properties of a view mechanism determines which schema modifications can be simulated. For instance, the models presented in [11, 34] allow to simulate the addition of a new attribute to a class because they consider views which can include non-derived additional attributes. However, the view model introduced in [27] supports a very limited number of changes because it does not support neither the generation of persistent identifiers for view instances nor views augmenting class definitions. Views allow the definition of external schemas with the meaning proposed by the ANSI three-level architecture: each external schema consists of a set of views and base classes specifying how a user perceives the database. Some of the proposed object-oriented view models [29, 42, 39] have considered external schemas. In those models view definitions are part of a schema view.

The design of our view model has been influenced by two main objectives: (*i*) it must be sufficiently powerful for supporting all the kinds of schema changes included in well-known taxonomies [6, 32], and (*ii*) it must support the definition of external schemas whose properties are identical to those of a base schema, so that it is possible to develop object-oriented applications on an external schema. The second goal implies that classes and views must have the same nature. In order to satisfy these requirements we have introduced the concept of *schema view*. A schema view encapsulates a set of related view definitions, with a well-defined purpose, such as to define an external schema or to perform some schema changes. The database administrator initially defines a base schema on which the object database is created. A schema view can be derived from either the base schema or from another schema view.

3.2. View placement

A relational schema consists of a set of relations, while an object-oriented schema consists of a class hierarchy, being classes connected by inheritance relationships. Therefore, an object-oriented view model must deal with the problem of inserting views in the class hierarchy. We refer to this problem as *view placement problem*. Three kinds of solutions have been proposed to solve it:

1. Views are automatically positioned in the class hierarchy by an integration algorithm [1, 37, 39].
2. The user explicitly specifies the position of the view in the class hierarchy [23].
3. The view and class hierarchies are kept separated, that is, a class can inherit only from classes and a view can inherit only from views. By contrast, a *view derivation relationship* relates a base class and a view; the semantic of this relationship is "the view is derived from the base class" [8].

Bearing in mind that views must behave like classes, the integration of views and classes in a unique schema may appear the most appropriate solution. However, solution 1 above has two problems: first, the problem of integrating a view in an existing schema is in general undecidable [7, 39]; second, the hierarchy may become very large because of many intermediate classes that are not semantically meaningful. With the second solution the mix of classes and views in the same hierarchy may cause confusion in the user. Moreover, checks must be made by the system to ensure that the specified position is coherent with the definition of the view. Note, moreover, that when one wants to support the mingling of classes and views in a single inheritance hierarchy the placement of a view in the inheritance hierarchy must be made by considering the two aspects of a view definition: the signature (list of attributes and methods, along with their domains) and the extent of the view (the set of instances that will be materialized when the query part of the view is evaluated). It may happen that the type of a view is a supertype of the type of the class from which it is derived but its extent is a subset of the extent of that class. Consider as an example a view extracting (that is, selecting) some objects from a class and projecting out some of their attributes. In a well-formed inheritance hierarchy, subtyping and set containment between class extents go together, since the extent of a class is defined as a subset of the extent of its superclass(es). Finally, note that none of the models taking the approach of inserting views in the class hierarchy supports object-generating views[9].

Therefore, we think that the more adequate approach is the separation of classes and views in different hierarchies, extending the object-oriented schema with the *view derivation relationship*, because it leads to a schema easier to understand. A view can be derived from other views, but in any case it will always be connected to base classes. With respect to the view derivation relationship, the term *root class* always refers to classes or views from which a view is derived. Let v be a view derived from the root class c, then the view derivation relationship denotes that v *is a view of* c, or, equivalently, that c *is a root class of* v.

In order to satisfy the above mentioned goals, it is also necessary to introduce a *view inheritance relationship*, similar to the class inheritance relationship existing in the base schema. The inheritance relationship is orthogonal to the view derivation relationship. The view inheritance relationship organizes views in an ISA hierarchy similar to the class hierarchy. An important difference between class and view inheritance is the fact that unlike a class, a view is not explicitly populated; rather its population is derived from the population of its root classes by the view query. Thus, to ensure that the instances of a view are a subset of the instances of its superviews, we impose two restrictions: (*i*) a view v_1 can be declared subview of a view v_2 if and only if the root class of v_1 is a direct or indirect subclass of the root class of v_2, and (*ii*) the query of a view must be stronger than the view queries of its superviews. This topic is dealt with in Subsection 5.3. Like the second solution, the user is responsible for the declarations of inheritance relationship, but the system must ensure that the placement of a view does not violate the semantics of a well-formed view inheritance hierarchy. Recently, two view models have been proposed which also solve the view placement problem by combining the view derivation and view inheritance relationships: the extension presented in [42, 41] to the view model described in [1] for the O_2 system, and the view mechanism for the UniSQL system [27].

3.3. *Object-generating vs object-preserving views*

If views are to be used as classes, it is essential that their instances are objects, that is, that they are provided with persistent identifiers. In most situations there is indeed the need of referencing view instances. Two distinct kinds of views can however be identified:

- *object-preserving views*: they are views that only extract objects from existing classes; the instances of these views can be identified by the identifiers of the extracted base objects.
- *object-generating views*: they are views creating new objects; the instances of these views must be identified by newly generated object identifiers.

Most of the approaches [31, 43, 40, 38] only consider object-preserving views. Thus, views only provide different views of existing objects. This approach is particularly useful for supporting objects with multiple interfaces and context-dependent behavior. In this sense, object-preserving views are similar to *roles* [3, 20, 35]. However, this kind of views is not powerful enough for supporting all kinds of database reorganization. In [33] a model is described whose views are object-preserving, but with the capability of including new non-derived attributes from existing data. By contrast, object-generating views are proposed in [8] for supporting schema evolution, so that the evaluation of a query always returns new objects. Object-generating views are also considered in [23], where a query can include an *oid function*: a partial function indicating that the query returns a set of objects whose identifiers are generated by applying the function on the object identifiers assigned to its argument variables.

The approach proposed in [1] supports object-preserving views as well as value-generating views. Thus, two kinds of views are considered: *virtual classes*, populated by objects selected from already existing classes, and *imaginary classes*, populated by tuples for which new oids are generated. Virtual classes are defined by specialization or generalization of base classes, while imaginary classes are declared by queries that return sets of values. In this approach, an imaginary class C is populated by a query that returns a set of tuples. To each tuple t an oid denoted as $C(t)$ is assigned, using a function associated with the class that is applied to the tuple. Thus, in this approach queries create relations rather than creating sets of objects. Therefore, queries cannot be used to define views, since it is necessary to convert tuples into objects outside the query language. This object "creation" is performed at each view evaluation, thus the problem arises of assigning the same oid to the same tuple at each evaluation.

In our model, we consider both object-preserving and object-generating views. Thus, when the view query evaluation involves the creation of view instances in order to populate the view class (e.g. a join operation), new oids are generated. By contrast, when views are populated by extracting existing objects from a class, possibly, modifying their structure and behavior, (e.g. a selection or projection operation) the view instances preserve the identifiers of base objects, instead of generating new objects. The support of object-preserving views requires that an object instance of a base class can also be an instance of all those views whose query is satisfied by the object. Thus, the use of the same identifier for denoting an object which is instance of both a class and a view implies that references to this object can only be solved by taking into account the *context* of the reference. However, this is already the case in Chimera, because the language supports objects belonging to multiple most specific classes [9]. We will elaborate further on the problem of solving object references in Subsection 6.3.

Together with views generating objects, we also allow a user to specify that the instances of a view are *not* objects, and thus are not provided with persistent identifiers. Therefore persistent identifiers are generated only when needed, that is, only when the view must be used as a class. Views whose instances are values rather than objects (which we refer to as *set-tuple views*) are useful to include relations in the object data model , thus providing a form of downward compatibility with respect to the relational model. However, views, whose instances are not provided with persistent identifiers, can only be used as shorthand in queries and cannot have additional attributes. As a default, we assume the view query returns a set of persistent objects, so that the view has an extension defined as the set of the oids of objects that belong to the view (both for object-generating and object-preserving views). Whereas the user needs to specify whether a view generates persistent identifiers or not, the system will be able to check whether a view preserves or generates objects, by analyzing the view query, as we will see later in Section 4.

3.4. Discussion

Table 1 compares the most relevant proposals for object-oriented view mechanisms, by taking into account the dimensions dealt with throughout the section. The table shows that our view model aims at supporting both external schemas and schema evolution. Moreover, we choose to support both object-preserving and object-generating views. As far as the view placement problem is concerned, in our approach we separate the class and the view hierarchies. A view is linked to its root classes by a *view derivation* relationship which is orthogonal to the inheritance relationship among classes. Views are moreover related by an inheritance relationship that results in a ISA hierarchy, just as classes. A schema view is a set of related view definitions for a well-defined purpose (e.g. define an external schema, define a schema change, define an authorization unit); a view must be part of a schema view.

Finally, we have considered whether the model allows a view to add new non-derived attributes. A view may indeed hide, modify and add features to those of the classes (or views) it is derived from. All models allow to hide features, as well as to add new methods and to change the implementation of methods. The addition of non-derived attributes has only been considered in some models supporting schema evolution. This property, obviously, increases the number of possible schema changes, but on the other hand it makes the implementation more difficult. Our model has this augmentation capability, thus view instances can have an additional storage for new attributes. Of course, other actions (e.g. hide attributes, hide/add/modify methods) are also possible.

4. Chimera view definition language

In this section we describe the view definition language proposed for Chimera, designed according to the basic choices we have discussed in Section 3. In our model, a view is defined as a query on one or more base classes whose result is a new class, as a natural adaptation of relational views. A view is identified by a name, which is the proposed identifier for the view. A view definition is thus similar to a class definition (name, list of attributes, list of methods, list of constraints) except that it includes a query on one or more base classes determining the view population. A view can be used just like a class. For example, the domain of an attribute, parameter or variable of a view can be another view.

Throughout the paper, we use as a running example the base schema presented in Figure 1. The classes of the `FacultyLibrary` schema only contain instance attributes, the other features are excluded for the sake of simplicity. The symbol '*' denotes that

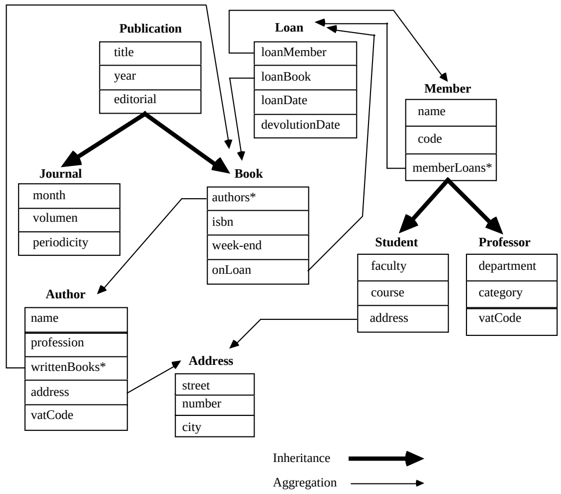

FIG. 1. `FacultyLibrary` example schema.

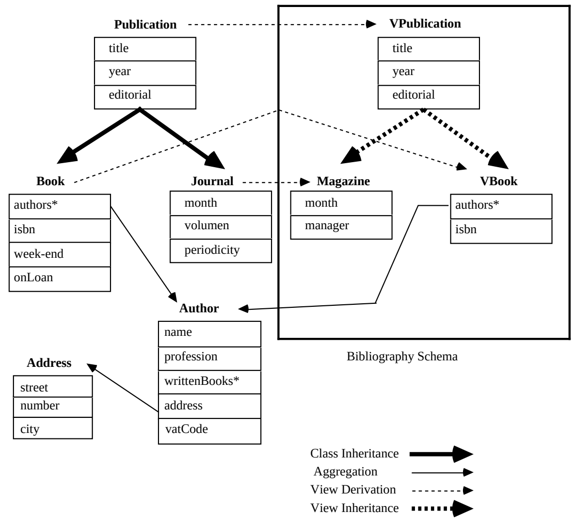

FIG. 2. Schema including views derived from the `FacultyLibrary` schema.

an instance attribute is multi-valued. Figure 2 shows a schema view named `Bibliography` derived from the `FacultyLibrary` base schema of our running example. This schema view illustrates how *view derivation* and *view inheritance* relationships can be used when views are derived from a given schema, in order to create a new schema. The `Bibliography` schema contains the views `Vpublication`, `Magazine` and `Vbook` which have been derived from the classes `Publication`, `Journal` and `Book`, respectively. In this example, the root classes are base classes because the view schema is derived from the base schema. The view `Magazine` has an additional attribute, `manager`, and hides the attributes `periodicity` and `volumen`; the view `Vbook` has no additional attributes and hides `week-end` and `onLoan`, while the view `Vpublication` imports all the attributes from

TABLE 1. Comparison of the Chimera view model with other OO view models

	[8]	[27]	[36]	[39]	[42]	Our model
data model	general	UniSQL	MultiView	COCOON	O_2	Chimera
query language	object-oriented predicate calculus	Object SQL	Object algebra	Object algebra	O_2	deductive rules
external schemas	NO	NO	YES	NO	YES	YES
schema evolution	YES	L.F.	L.F.	L.F.	L.F.	YES
view integration problem	separate view hierarchy	separate view hierarchy	views inserted in class hierarchy	views inserted in class hierarchy[a]	separate view hierarchy	separate view hierarchy
object-preserving/ object-generating	generating	no persistent identifiers	preserving	preserving	both[b]	both
augment class definition	YES	NO	YES	NO	NO	YES

Legenda: L.F. limited form

[a] For operator individuals, it is specified how the view is inserted in the class hierarchy, but the classification of the views resulting of composite queries is not addressed.

[b] Actually, a query may return a set of tuples that are converted to new objects outside the query language.

```
VIEW SIGNATURE ViewName
    FROM RootClasses
    IMPORTED-FEATURES
        ATTRIBUTES ListOfImpAttrib
        OPERATIONS ListOfImpOper
        CONSTRAINTS ListOfImpCons
        C-ATTRIBUTES ListOfImpCattrib
        C-OPERATIONS ListOfImpCoper
        C-CONSTRAINTS ListOfImpCconst
    ADDITIONAL-FEATURES
        ATTRIBUTES ListOfAddAttrib
        OPERATIONS ListOfAddOper
        CONSTRAINTS ListOfAddConst
        C-ATTRIBUTES ListOfAddCattrib
        C-OPERATIONS ListOfAddCOper
        C-CONSTRAINTS ListOfAddCconst
    VIEW-QUERY SetOfDeductiveRules
    SUPERVIEWS ListOfViews
    OID bool
```

FIG. 3. View specification statement

the class `Publication` and does not add new attributes. For the remaining features, we suppose that the views import all features from their root classes and do not augment class definitions. The views `Magazine` and `Vbook` have been declared subviews of the view `Vpublication`. Note that the view `Vpublication` is the identity view[10] of the class `Publication`, being intended to have a hierarchy of publications in the schema view. As remarked in [39], schema views must satisfy some consistency constraints concerning the schema closure. For instance, because the domain of the attribute `authors*` is the class `Author`, the closure of the schema view `Bibliography` must contain the identity views of classes `Author` and `Address`. The closure of schema views will be discussed in Subsection 6.1.

As for Chimera classes, we can distinguish two components in a view definition: *specification* and *implementation*. They are dealt with in the following subsections.

4.1. View specification

The format of a view definition statement is shown in Figure 3. The clauses of the view definition statement in Figure 3 have the following meaning:

- `ViewName` denotes the view name and must be distinct from the names of all existing views and classes.
- The `FROM` clause lists the root classes (which can be either classes or views) from which the view is derived.
- The `IMPORTED-FEATURES` and `ADDITIONAL-FEATURES` clauses specify the view features, distinguishing between imported and additional ones. They are discussed in detail in Subsection 4.1.2..
- The `VIEW-QUERY` clause specifies the population of the view, by means of a set of Chimera deductive rules. It is discussed in detail in Subsection 4.1.1..
- The `SUPERVIEWS` clause declares the superviews of the view which is being defined, in a similar way as for classes. In Subsection 5.3 we will further discuss the meaning of this relationship.
- The `OID` clause contains a boolean value, indicating whether persistent object identifiers are provided for view instances. The default value is `true`. The `false` value is used for views whose instances are not provided with persistent identifiers, being thus values rather than objects (set-tuple views).

Some view definition statements are presented in Examples 1 and 2. They use the `FacultyLibrary` schema of Figure 1.

Example 1. *Suppose we wish to consider a class representing magazines whose periodicity is weekly. We can derive the* `Magazine` *view from the* `Journal` *class, populated with all the objects from* `Journal` *class such that* `periodicity = 'weekly'`. *The view has an additional attribute representing the magazine editor while the* `volumen` *and* `periodicity` *attributes are hidden. The* `VPublication` *view is declared as a superview of the view* `Magazine`.

```
VIEW SIGNATURE Magazine
     FROM Journal
     IMPORTED-FEATURES
        ATTRIBUTES -volumen, -periodicity
     ADDITIONAL-FEATURES
        ATTRIBUTES manager: string
        OPERATIONS changeManager(in NewMan:string)
     VIEW-QUERY Magazine(X) ← Journal(X),
                X.periodicity = 'weekly'
     SUPERVIEWS VPublication
     OID true
```

◇

Example 2. *Suppose we wish to define a class containing all professors which are authors of some book published by the* `University Editorial` *and available in the library. We can define the view* `ProfAuthor` *as a view-query expressing an explicit join among the classes* `Professor` *and* `Author`, *through the* `vatCode` *attribute included in the two classes.*

```
VIEW SIGNATURE ProfAuthor
     FROM Author, Professor
     IMPORTED-FEATURES
        ATTRIBUTES name of Author,
                vatCode of Author,
                bookTitles: set-of(string) derived,
                city: author.address.city
     ADDITIONAL-FEATURES
        OPERATIONS changecity(in City: string)
     VIEW-QUERY ProfAuthor(X) ← Author(Y),
                Professor(Z), Book(W),
                Y.vatCode = Z.vatCode,
                W in Y.writtenBooks,
                W.editorial = 'University Editorial'
     OID true
```

◇

4.1.1. View query Whereas most view models express the view query by an object algebra or calculus, we use Chimera deductive rules. The `VIEW-QUERY` clause contains one or more Chimera deductive rules specifying the view population. The head of these rules is a class formula on the name of the view whose population is being defined and the body is an arbitrary formula on instances of the root classes. We remark that only side-effect free methods are allowed in queries. From now on, the *view-query* term will denote the collection of rules that define the view population. The view-query defines the extension of the view, while the structure of each instance belonging to this extension is based on imported and additional attributes.

Example 3. *The following query defines the population of a view* `CsStudent` *derived from the class* `Student` *of the schema* `FacultyLibrary`, *retrieving the students whose faculty is* `Computer Science` *and that have borrowed at least a book.*

```
CsStudent(X) ← Student(X), Book(Z),
               X.faculty = 'Computer Science',
               Z in X.memberLoans
```

◇

If the variable of the class formula in the head of the rule appears in (a class formula of) the rule body, then the view is an object-preserving view (examples of object-preserving views are the views of Example 1 and of Example 3), otherwise the view is an object-generating view (an example of object-generating view is the view of Example 2).

For object-preserving views, if the class formula in the rule body which contains the variable in the head of the rule is on the root class c, then the view instances are objects extracted from c. Thus, all the members of c that satisfy the body of the rule are instances of the view. For example, instances of the `CsStudent` view of Example 3 are objects belonging to class `Student`. By contrast, the query of an object-generating view returns a set of base object tuples. For each tuple in this set a new object is generated and added to the view extension, and the correspondence between the new object identifier and the base object identifier is stored in a persistent table, named DERIVED BY. It is sometimes useful, in the definition of the implementation for derived imported attributes, to explicitly refer to the base objects a certain view object has been derived from. We thus extend the syntax of Chimera deductive rules with a special atomic formula built using the ternary predicate `derived-by`, whose first argument is the identifier of a view instance, second argument is a class identifier and third argument is an object identifier. The third argument is bound to the base object, instance of the specified class, from which the specified view object has been derived. This predicate is simply a mean to refer to the DERIVED BY table from the body of a deductive rule.

If the instances of a view are newly generated objects, the view-query of such a view contains a variable in the head which is not contained in any atomic formula in the body. In such cases, we may think that an atomic formula `next-oid(X)` is automatically added to the rule body, being `X` the variable appearing as argument of the class formula in the rule head, denoting the newly generated object. In Example 2 the body of the view-query will include `next-oid(X)`, being the rule head the class formula `ProfAuthor(X)`. The meaning of atomic formulas built with the `next-oid` predicate is the following:

1. a new object identifier must be generated for each successful evaluation of the formula in the rule body;
2. the generated identifier must be added to the view extension;
3. for each new generated oid, an entry containing the generated oid and the oids of base objects from which it has been derived, must be stored in the DERIVED BY table.

Note that a view-query may consist of several deductive rules. Thus, our view language is able to express views defined by queries on a union of classes using alternative predicates [8], as shown by the following example. The example also shows that it is possible to define an object-preserving view by extracting objects from two or more root classes.

Example 4. The following query retrieves all the members that have borrowed at least one book, such that, if they are student, their faculty is `Computer Science` *and if they are professors, their category is* `full time`.

```
LoanMembers(X) ← Student(X), Book(Z),
                 Z in X.memberLoans,
                 X.faculty = 'Computer Science'
LoanMembers(X) ← Professor(X), Book(Z),
                 Z in X.memberLoans,
                 X.category = 'full time'
```

◇

Finally, we remark that at the time being we do not consider recursive views. A view v can be defined in terms of another view v' provided that v' is not defined, directly or indirectly, in terms of v. Thus, each view can be ultimately seen as defined in terms of base classes.

4.1.2. Imported and additional features Besides specifying the root classes and the query defining the view, the view signature also specifies information on view features. We distinguish between *imported* and *additional* view features: imported features are obtained from one of the root classes, while additional features are explicitly defined for the view. When considering attributes, however, a view can have some attributes not belonging to the signature of any root class, but whose value can be derived (that is, computed) starting from the values of some attributes in the root classes. We consider this kind of attribute as imported rather than as additional, to remark that for additional attributes new storage space must be allocated and a value must be provided for each view object, since all additional attributes have a null value upon view materialization.

In the `IMPORTED-FEATURES` clause of the view definition statement, `ListOfImpOper`, `ListOfImp-Const`, `ListOfImpCattrib`, `ListOfCoper` and `ListOfImpCconst` denote the lists of features imported from root classes. In the case of imported attributes, `ListOfImpAttrib` specifies which attributes among the ones of the base objects retrieved by the query are part of the view instances. For each clause the associated list contains one or more items specifying which features are imported. There are different options to specify the imported features: listing the features to be imported, specifying that all the features (of a root class) are imported, specifying that a feature is hidden, or specifying that a feature is renamed in the view. The option of specifying which features (of a given root classes) are hidden is useful when the number of imported features is greater than the number of hidden ones; another important (semantic) advantage of allowing the specification of hidden features rather than requiring the specification of imported ones, is that the view may change if the root class changes, e.g. if new attributes are added to the root class they are added to the view in the former case, whereas they are not added in the latter. In the case of imported attributes, there are two additional formats for introducing derived attributes not corresponding to any attribute of the root classes, but whose value can be computed from attributes in the root classes by means of Chimera deductive rules. Those attributes are part of the state of the instances of the view but do not belong to the state of any base object the view object is derived by. A first option is to indicate that a view attribute is derived; its implementation must be given in the view implementation (see Example 5 below). The derived imported attribute `bookTitles` in the view of Example 2 could be implemented by the following deductive rule:

```
X in self.bookTitles ← Author(Y), Book(W),
    derived-by(self, Author, Y),
    W.editorial = 'University Editorial',
    Y in W.authors
```

The above option supports the redefinition of the domain of an attribute imported from a root class. It may be useful for restricting the domain of an attribute to a subtype of the current attribute type, or for changing it to a view derived from the class which is the attribute domain.

A second option is to specify that a view attribute corresponds to a nested attribute of one of the root classes of the considered view. Example 2 shows the specification of a view called `ProfAuthor` where the imported attribute `city` is defined through the path expression `Author.address.city`. This expression specifies that the view `ProfAuthor` has an attribute named `city` whose value, for each object o belonging to the view, is the value of `address.city` in the base object from root class `Author` by which o is derived[11].

The various formats for importing features in view specification are specified in Appendix A.

In the `ADDITIONAL-FEATURES` clause of the view definition statement, `ListOfAddAttrib`, `ListOfAddOper`, `ListOfAddConst`, `ListOfAddCattrib`, `ListOfAddCoper` and `ListOfAddCconst` denote lists of additional features specifications, one list of signatures for each kind of feature. A feature signature is expressed exactly like in class signatures.

4.2. View implementation

The view *implementation* is exactly like a class implementation, including the set of Chimera deductive rules that specify the implementation of the imported and additional derived attributes, methods and constraints. The definition of a view implementation has the format presented in Figure 4.

Example 5. The implementation of the view `ProfAuthor` *of Example 2 may be specified by the following statement.*

```
VIEW IMPLEMENTATION ProfAuthor
    ATTRIBUTES
       X in self.bookTitles ← Author(Y), Book(W),
            derived-by(self, Author, Y),
            W.editorial = 'University Editorial',
            Y in W.authors
    OPERATIONS
       changecity(City):
            → modify(ProfAuthor.city,Self,City)
```

◇

We remark that the view implementation is expressed by making use of Chimera deductive rules, as defined in Section 2, extended as follows:

- the rules specifying the view population may contain a variable in the head that do not appear in any atomic formulas in the bodies; this is the format for specifying object-generating views and it is handled, as seen in Subsection 4.1.1., by inserting in the rule body a special atomic formula on the `next-oid` predicate;
- the rules specifying the implementation for derived attributes may contain, in their body, atomic formulas on the ternary predicate `derived-by`, which allows to refer to the base object(s) from which the view object at hand has been derived.

```
VIEW IMPLEMENTATION ViewName
    ATTRIBUTES derived attribute implementation
    OPERATIONS operation implementation
    CONSTRAINTS constraint implementation
    C-ATTRIBUTES derived c-attribute implementation
    C-OPERATIONS c-operation implementation
    C-CONSTRAINTS c-constraint implementation
```

FIG. 4. View implementation statement

5. Formal definition of the Chimera view model

In this section, the view model proposed for Chimera is formally defined. First of all, we define the notion of view and we discuss what view instances are. Then, a subview relationship is defined, showing how views can be part of view hierarchies like classes are part of class hierarchies.

In the following, let $\mathcal{OI}$ denote a set of object identifiers and $\mathcal{CI}$ denote a set of class identifiers. Moreover, we consider a set of type names $\mathcal{TN}$, a set of attribute names $\mathcal{AN}$, a set of method names $\mathcal{MN}$ and a set of constraint predicate symbols $\mathcal{PN}$. Finally, $\mathcal{V}$ denotes the set of Chimera values, defined starting from basic values and object identifiers and applying the set, list and record constructors [21]. From now on, we make use of the dot notation to refer to the components of a tuple: $t.c$ denotes the component of the tuple t named c.

5.1. Views

As for classes, we consider two components in a view: *signature* and *implementation*, which derive from the *specification* and *implementation* components in the view definition statement, respectively.

5.1.1. View signature In the following, let $\mathcal{VI}$ denote a set of view identifiers, and $\mathcal{CVI}$ denote the set of class and view identifiers, thus, $\mathcal{CVI} = \mathcal{VI} \cup \mathcal{CI}$. Furthermore, in order to define how the signature of a view is obtained from the view definition statement, given a view definition V, we define the following structures:

- $iStruct(V)$: it is obtained from the lists `listOfImpAttrib` and `listOfImpCattrib`. It contains the information on attributes and c-attributes imported from the root classes. It is a record value with two sets named *inst* (including instance attributes) and *class* (including class attributes), whose items are triples

  $$(a\_name, a\_dom, a\_st)$$

  where

  – $a\_name \in \mathcal{AN}$ is the attribute name

– $a\_dom \in \mathcal{T}$ is the attribute domain

– $a\_st \in \{ext, der\}$ is the attribute type, that is, whether it is extensional or derived.

- $iBeh(V)$: it is obtained from the lists `listOfImpOper` and `listOfImpCoper`. It contains the information on methods and c-methods imported from the root classes. It is a record value with two sets named *inst* (including instance methods) and *class* (including class methods), whose items are pairs

$$(op\_name, op\_sign)$$

where

– $op\_name \in \mathcal{MN}$ is the method name

– op_sign is the signature of the method, expressed as

$$T_1 \times \ldots \times T_k \rightarrow T$$

with $T_1, \ldots, T_k$ and T types in $\mathcal{T}$, representing, respectively, domains of input and output parameters of the method.

- $iConst(V)$: it is obtained from the lists `listOfImpConst` and `listOfImpCconst`. It contains the information on constraints and c-constraints imported from the root classes. It is a record value with two sets named *inst* (including instance constraints) and *class* (including class constraints), whose items are pairs

$$(con\_name, con\_sign)$$

where

– $con\_name \in \mathcal{PN}$ is the constraint name

– con_sign is the signature of the constraint, expressed as

$$T_1 \times \ldots \times T_k$$

with $T_1, \ldots, T_k$ types in $\mathcal{T}$, representing domains of the output parameters of the constraint.

- $aStruct(V)$: it is obtained from the lists `listOfAddAttrib` and `listOfAddCattrib`. It is exactly like *iStruct*, but it contains information on additional attributes instead of on imported ones.
- $aBeh(V)$: it is obtained from the lists `listOfAddOper` and `listOfAddCoper`. It is exactly like *iBeh*, but it contains information on additional methods instead of on imported ones.
- $aConst(V)$: it is obtained from the lists `listOfAddCconst` and `listOfAddCconst`. It is exactly like *iConst*, but it contains information on additional constraints instead of on imported ones.

Table A1 in Appendix A specifies how the introduced structures are obtained from the corresponding fields in the view specification statement.

Definition 4 (View Signature). *Given a view specification V, the corresponding view signature is a tuple*

$$VC = (id, struct, beh, constr, state, mc, q)$$

generated as follows:

- $id \in \mathcal{VI}$ is the view identifier specified in the view definition;
- $struct = aStruct(V).inst \cup iStruct(V).inst$;
- $beh = aBeh(V).inst \cup iBeh(V).inst$;
- $constr = aConst(V).inst \cup iConst(V).inst$;
- *state* is a record value, containing the values for the view attributes which are obtained from $aStruct(V).class \cup iStruct(V).class$; two additional fields belong to the record, *extent* and *proper_extent*;
- *mc* is the identifier of a virtual metaclass corresponding to the view. This identifier can be any name not used as name of other class or metaclass. The metaclass is derived as follows:

  – $struct = aStruct.class(V) \cup iStruct.class(V) \cup \{(proper\_extent, set\text{-}of(id), extensional), (extent, set\text{-}of(id), extensional)\}$,

  – $beh = aBeh.class(V) \cup iBeh.class(V)$,

  – $constr = aConst.class(V) \cup iConst.class(V)$;

- q is the view-query, that is, a set of deductive rules specifying the view population; this set is exactly the one specified in the `VIEW-QUERY` clause of the view definition statement. □

The attributes *extent* and *proper_extent* in the view state denote respectively the set of all the oids of objects members of the view and the oids of objects instances of the view. Therefore the *proper_extent* field of the view state contains the set of objects belonging to the view and not belonging to any of its subview in the view inheritance hierarchy.

Example 6. Referring to the view `ProfAuthor` *specified in Example 2, the corresponding view signature is as follows. Let { (name, string, ext), (vatcode, integer, ext) } be included in the struct component of the class identified by* `Author`. *Let E and PE denote two sets of (view) object identifiers such that $PE \subseteq E$. Then:*

$VC.id =$ `ProfAuthor`
$VC.struct = \{ (name, string, ext), (vatcode, integer, ext), (bookTitles, set\text{-}of(string), der), (city, string, der)\}$
$VC.beh = \{(changecity, string \rightarrow ProfAuthor)\}$
$VC.constr = \emptyset$
$VC.state = (extent : E, proper\_extent : PE)$
$VC.mc =$ `MProfAuthor`

```
VC.q = ProfAuthor(X) ← Author(Y), Professor(Z),
        Book(W), Y.vatCode = Z.vatCode,
        W in Y.writtenBooks,
        W.editorial = 'University Editorial'
```

◇

The identifier of a view VC denotes the object type corresponding to VC. Such object type is the type of the identifiers of the objects instances of the view. A value type is moreover implicitly associated with each view, representing the type of values that constitute the state of the view instances. If the *struct* component of a view VC is the set $\{(a_1, T_1, at_1), \ldots, (a_n, T_n, at_n)\}$, each object instance of VC must have as state a value of (record) type $record\text{-}of(a_1 : T_1, \ldots, a_n : T_n)$. This type, which describes the structure of the objects instances of the view, is the *structural type* of the view, and it is denoted by $stype(v)$, being $v \in \mathcal{VI}$ the view identifier of VC ($VC.id = v$). Two components can be distinguished in the state of a view instance: an additional component and an imported component. Therefore, given the view structural type $stype(v) = record\text{-}of(a_1 : T_1, \ldots, a_n : T_n)$, it is possible to partition this record type in two record types:

- $stype\_imp(v) = record\text{-}of(a_1 : T_1, \ldots, a_k : T_k)$, and
- $stype\_add(v) = record\text{-}of(a_{k+1} : T_{k+1}, \ldots, a_n : T_n)$, for a $k \leq n$

where we assume that $iStruct(V).inst = \{(a_1, T_1, at_1), \ldots, (a_k, T_k, at_k)\}$ and $aStruct(V).inst = \{(a_{k+1}, T_{k+1}, at_{k+1}), \ldots, (a_n, T_n, at_n)\}$. Therefore, $stype\_add(v)$ denotes the additional structural type while $stype\_imp(v)$ denotes the imported structural type. We distinguish these two components in the structural component of a view, because the additional component is related to additional attributes that must be stored for view instances, whereas the imported component is related to attributes imported from root classes. The values of the imported component are, therefore, retrieved (or computed) starting from values already stored in the database. The system does not allocate space for imported attributes, while it does for additional ones.

The set of the classes[12] from which a view is derived can be represented as a function $defined\_on : \mathcal{VI} \rightarrow 2^{\mathcal{CVI}}$. This function returns the elements of the `RootClasses` list in the `FROM` clause of the view definition statement. For example, referring to Example 2, $defined\_on(\texttt{ProfAuthor}) = \{\texttt{Professor}, \texttt{Author}\}$. Note that, since views can be defined in terms of other views, the function *defined_on* returns a set containing base classes as well as views. It is however possible to determine the base classes on which a view is defined, by recursively applying function *defined_on*. For that purpose the function $defined\_on^* : \mathcal{VI} \rightarrow 2^{\mathcal{CI}}$, defined as follows, can be used: $defined\_on^*(v) = defined\_on(v)$ if $defined\_on(v) \subseteq \mathcal{CI}$, whereas $defined\_on^*(v) = \{c_1, \ldots, c_k\} \cup defined\_on^*(v_1) \cup \ldots \cup defined\_on^*(v_h)$ if $defined\_on(v) = \{c_1, \ldots, c_k, v_1, \ldots, v_h\}$, $\{c_1, \ldots, c_k\} \subseteq \mathcal{CI}$ and $\{v_1, \ldots, v_h\} \subseteq \mathcal{VI}$.

We represent through a boolean function $oid : \mathcal{VI} \rightarrow Bool$ whether or not the view instances are provided with persistent identifiers, according to the boolean value specified in the `OID` clause of the view definition statement. Moreover, we represent through a boolean function $new\_oid : \mathcal{VI} \rightarrow Bool$ whether the view is object-generating or object-preserving. That is, $new\_oid(v) = true$ for object-generating views, while it is $false$ for object-preserving ones.

5.1.2. View implementation

A view implementation consists of three sets of rules:

1. a set of deductive rules specifying the implementations of the view derived attributes; an implementation must be provided for
   - each additional derived attribute; in this case the implementation is specified in the implementation part of the view definition;
   - each imported attribute declared as derived in the signature part of the view definition; in this case the implementation is specified in the implementation part of the view definition;
   - each imported attribute which is derived in the root class from which it is taken; in this case the implementation is the same as the one in the (implicitly or explicitly) referred class;
   - each imported attribute declared as $\mathsf{a} : \mathsf{c.a_1.\cdots.a_n}$ in the signature part of the view definition; in this case the implementation consists of

     a. the rule
     $$self.a = X \leftarrow c(Y), Y = self, X = Y.a_1.\cdots.a_n$$
     if $new\_oid(v) = false$;

     b. the rule
     $$self.a = X \leftarrow c(Y), derived\_by(self, c, Y), X = Y.a_1.\cdots.a_n$$
     if $new\_oid(v) = true$;
2. a set of deductive rules specifying the constraints on the view population; this set consists of the rules specified in the view class implementation for additional constraints and the rules in the intended root class implementation for imported constraints;
3. a set of update rules specifying the implementation of the view operations; this set consists of the rules specified in the view class implementation for additional operations and the rules in the intended root class implementation for imported operations.

Definition 5 (View Implementation). *Given a Chimera view signature*
$$VC = (id, struct, beh, constr, state, mc)$$
an implementation for VC consists of a set of deductive rules, specifying

- an attribute implementation for each derived attribute in *struct*;
- a constraint implementation for each constraint in *constr*;
- an operation implementation for each operation in *beh*. □

Example 7. The implementation of the view of Example 6 consists of the following rules.

```
X in self.bookTitles ← Author(Y), Book(W),
     derived-by(self, Author, Y),
     W.editorial = 'University Editorial',
     Y in W.authors

self.city = X ← Author(Y),
     derived-by(self, Author, Y),
     Y.address.city = X

changecity(City):
     → modify(ProfAuthor.city,Self,City).
```

◇

5.2. View instances

In our approach, the evaluation of a view definition results in a view whose instances have a structure defined by function *stype* introduced above. Thus, a view is a class and, if the value specified in the `OID` clause of the view definition statement is `true`, its instances are objects referred through immutable identifiers. The extent of a view may consist of objects extracted from an existing class or view (for the views v such that $new\_oid(v) = false$), or it may consist of newly generated objects (for the views v such that $new\_oid(v) = true$). In the first case, the extracted object is, obviously, an instance of the class from which it has been extracted and it is also an instance of the view. Thus, the object belongs to multiple most specific classes. We have addressed the problem of objects belonging to multiple most specific classes in [9] where only base classes are considered. Here, we extend that approach by considering also views. Thus, an object may belong to several most specific classes and to a set of views derived from them. The notion of object can be formalized as follows.

Definition 6 (Object). *An object is a tuple*

$$o = (i, v, VS)$$

where:

- $i \in \mathcal{OI}$ is the identifier of o;
- $v \in \mathcal{V}$ is a value, called state of o;
- $VS \subseteq \mathcal{CVI}$ is the set of most specific classes and views to which o belongs. □

For object-generating views, a new persistent oid is generated for each view instance, in the same way as base object identifiers are generated upon object creation. The objects generated by the view evaluation have only a virtual nature (though part of their state, that is, the values for additional attributes, is stored) and are referred to as *view objects*. They are objects according to Definition 6, though they do not belong to any base class. We may thus partition the set $\mathcal{OI}$ of object identifiers in two sets: $\mathcal{BOI}$, the set of base object identifiers; and $\mathcal{VOI}$, the set of view object identifiers, that is, the set of identifiers corresponding to view objects, generated upon view materialization. An object o has an identifier belonging to $\mathcal{VOI}$ if it has only a virtual nature, that is, if it is not an instance of any base classes.

Definition 7 (View Object). *A view object is an object o defined according to Definition 6 such that*

- $o.i \in \mathcal{VOI}$, and
- $o.VS \subseteq \mathcal{VI}$. □

We remark that $o.i \in \mathcal{VOI} \Rightarrow o.VS \subseteq \mathcal{VI}$, that is, a view object belongs only to views. By contrast, $o.i \in \mathcal{BOI} \Rightarrow \exists c \in o.VS$ such that $c \in \mathcal{CI}$, that is, a base object belongs to at least one base class.

The function $derived\_by : \mathcal{VOI} \rightarrow 2^{\mathcal{BOI}}$, for each view object identifier $i \in \mathcal{VOI}$, returns the set of identifiers of base objects from which the view object identified by i has been derived. This function is defined by recursively replacing each view object identifier i with the set of identifiers appearing in the columns of the Derived By table row, whose first column contains i. This process ends when the set contains only base object identifiers. If, given a view identifier v, $defined\_on^*(v) = \{c_1, \ldots, c_n\}$ then, for each view object identifier i such that $i \in [\![v]\!]$, $derived\_by(i) = \{i_1, \ldots, i_n\}$ and for each $j, 1 \leq j \leq n$, a class c_k exists, $1 \leq k \leq n$, such that $i_j \in [\![c_k]\!]$.

5.2.1. Object state The state of an object belonging to several most specific classes (and views), should be a record value having as fields the union of all the attributes in those classes. However, the sets of attributes in the object most specific classes and views may be non-disjoint. To handle this situation we introduce the notion of *source* of an attribute. If an attribute belongs to the intersection of the attribute sets of two classes and it has in both classes the same source, then the attribute is semantically unique, and thus the object must have a unique value for this attribute. If, by contrast, the attribute has different sources, then the two attributes in the two classes (views) have accidentally the same name, but represent different information, that must be kept in separate ways. Thus, the object may have two different values for the two attributes (a renaming policy is applied).

We now specify the notion of *source* of an attribute. For base classes[13], the source of an attribute a in a class c is the most general superclass of c in which the attribute a is defined. Thus, it is the class from which c has inherited attribute a. Two base classes have a common attribute with the same source if they inherit it from a common superclass. For views, the source of an attribute can be:

- either the view itself, if the attribute is neither inherited nor included (with the meaning specified below) from any root class;
- the source of the attribute in the most general superview from which the view has inherited the attribute, for inherited attributes;
- the source of the attribute in the root class from which the view has taken the attribute, for included attributes.

In the second case, the most general superview from which the view has inherited the attribute is determined as for base classes [9]. In order to better explain the third case, consider the different formats for imported attributes presented in Appendix A. In cases a) and c) an attribute is included from a root class in the view, and the source is the class from which the attribute is taken (either the one explicitly specified or the only one containing that attribute). By contrast, in cases d), e) and f) the attribute is not actually included from the root class, and then its source is the view itself.

Let $\uplus$ be an operation defined as follows:

$$\begin{aligned}A(c_1) \uplus A(c_2) = &\{a \mid a \in A(c_1) \cup A(c_2) \\ &\quad \wedge a \notin A(c_1) \cap A(c_2)\} \cup \\ &\{a \mid a \in A(c_1) \cap A(c_2) \\ &\quad \wedge source(a, c_1) = source(a, c_2)\} \cup \\ &\{c_1\text{-}a \mid a \in A(c_1) \cap A(c_2) \\ &\quad \wedge source(a, c_1) \neq source(a, c_2)\} \cup \\ &\{c_2\text{-}a \mid a \in A(c_1) \cap A(c_2) \\ &\quad \wedge source(a, c_1) \neq source(a, c_2)\}\end{aligned}$$

where, given a class or view $c \in \mathcal{CVI}$, $A(c)$ denotes the set of attributes of that class and $source(a, c)$ denotes the source of an attribute a in a class c.

Let $dom(a, c)$, for $a \in A(c)$, denote the domain of attribute a in class c. Then, the state of an object (that is, $o.v$) is characterized by the set of its most specific classes and views (that is, $o.VS$) as follows.

Definition 8 [9]. *Let o be an object, such that $o.v = record\text{-}of(a_1 : v_1, \ldots, a_n : v_n)$. Then:*

- $\biguplus_{c \in o.VS} A(c) = \{a_1, \ldots, a_n\}$
- $\forall i, 1 \leq i \leq n, v_i \in \bigcap_{c \in o.VS} [\![dom(a_i, c)]\!]$. □

A function $value : \mathcal{OI} \times \mathcal{CVI} \rightarrow \mathcal{V}$ is defined that, given an object o and a class (or view) c, if o is a member of c (that is, if $o.i \in [\![c]\!]$), returns the state of object o seen as an instance of class c. That function only returns the fields of the state value proper of the structural component of c. Let $o = (i, v, VS)$ be an object and v be the record value $record\text{-}of(a_1 : v_1, \ldots, a_n : v_n)$. Then $value(i, c)$ is determined as follows, for $j = 1 \ldots n$:

- if $a_j \in A(c)$ then $a_j : v_j$ is a field of $value(i, c)$, and
- if $a_j = c\text{-}a'_j$ and $a'_j \in A(c)$, then $a'_j : v_j$ is a field of $value(i, c)$.

If an object $o = (i, v, VS)$ is a member of a class c, then $value(i, c)$ is uniquely determined and it is a legal value for the type $stype(c)$. This function is applied whenever we want to see an object as a view instance.

Since Chimera is a strongly typed database language, each object reference is assigned a single context in each expression. Thus, for each object reference we are able to determine (starting from the types declared for variables and using schema information) the class or view the referenced object must be seen an instance of. Note that this allows us to model notions such as context dependent access restriction and context dependent behavior, typical of data models including roles.

We remark that we have denoted as state the collection of all the attribute values of an object. Not all these values are stored, since some of them can be computed. In particular, derived attributes are not stored.

Example 8. Consider view `ProfAuthor`*, specified in Example 2, whose signature is given in Example 6. Then:*

$(i_1,\ record\text{-}of(name : \text{'}JohnSmith\text{'}, vatcode : 6432957,$
$\quad city : \text{'}NewYork\text{'}$
$\quad bookTitles : set\text{-}of(\text{'}Object\text{-}\text{Oriented Databases'})),$
$\{$ `ProfAuthor` $\}$)

with $i_1 \in \mathcal{VOI}$, is an example of object instance of the (object-generating) view `ProfAuthor`.

Consider now view `Magazine`, specified in Example 1, whose signature VC is such that

$VC.id =$ `Magazine`
$VC.struct = \{ (title, string, ext), (year, integer, ext),$
$\quad (editorial, string, ext), (month, string, ext),$
$\quad (manager, string, ext)\}.$

Then:

$(i_2,\ record\text{-}of(title : \text{'}InternationalJournal\text{'}, year : 1995,$
$\quad editorial : \text{'}ACM\text{'}, month : \text{'}April\text{'},$
$\quad manager : \text{'}AlanFord\text{'}, volume : 17,$
$\quad periodicity : \text{'}weekly\text{'}),$
$\{$ `Magazine, Journal` $\}$)

with $i_2 \in \mathcal{BOI}$, is an example of an object instance of the object-preserving view `Magazine`, and of the class `Journal`, whose signature C is such that $C.id =$ `Journal` and

$C.struct = \{ (title, string, ext), (year, integer, ext),$
$\quad (editorial, string, ext), (month, string, ext),$
$\quad (volume, integer, ext),$
$\quad (periodicity, string, ext) \}.$

Moreover,

$value(i_2,$ `Magazine`$) =$
$\quad record\text{-}of(title : \text{'}InternationalJournal\text{'},$
$\quad\quad year : 1995,$
$\quad\quad editorial : \text{'}ACM\text{'},$
$\quad\quad month : \text{'}April\text{'},$
$\quad\quad manager : \text{'}AlanFord\text{'}),$
$value(i_2,$ `Journal`$) =$
$\quad record\text{-}of(title : \text{'}InternationalJournal\text{'},$

$$
\begin{array}{l}
\quad\quad\quad year : 1995, \\
\quad\quad\quad editorial : \text{`}ACM\text{'}, \\
\quad\quad\quad month : \text{`}April\text{'}, \\
\quad\quad\quad volume : 17, \\
\quad\quad\quad periodicity : \text{`}weekly\text{'}) \\
value(i_2, \texttt{Publication}) = value(i_2, \texttt{VPublication}) = \\
\quad record\text{-}of(\ title : \text{`}InternationalJournal\text{'}, \\
\quad\quad\quad year : 1995, \\
\quad\quad\quad editorial : \text{`}ACM\text{'}).
\end{array}
$$

$\diamond$

5.2.2. Object consistency Each view object must be a consistent instance of all the views to which it belongs, exactly as each object must be a consistent instance of all the classes and the views to which it belongs. The following definitions formalize the notions of consistency we consider.

Definition 9 (Structural Consistency). *An object $o = (i, v, VS)$ is a structurally consistent instance of a class (or view) $c \in \mathcal{CVI}$ if v contains*[14] *a legal (record) value for the type $stype(c)$.* $\square$

Definition 10 (Constraint Consistency). *An object $o = (i, v, VS)$ is a constraint consistent instance of a class (or view) $c \in \mathcal{CVI}$, if o falsifies*[15] *all the bodies of rules implementing the constraints in the constr component of C, where $C.id$ is c.* $\square$

Definition 11 (View-query Consistency). *An object $o = (i, v, VS)$ is a view-query consistent instance of a view $c \in \mathcal{VI}$ if for a rule $H \leftarrow B$ in the view-query of c, o meets B.* $\square$

Definition 12 (Consistency). *An object $o = (i, v, VS)$ is a consistent instance of a base class $c \in \mathcal{CI}$ if o is both a structural and constraint consistent instance of c.*

An object $o = (i, v, VS)$ is a consistent instance of a view $c \in \mathcal{VI}$ if o, besides being both a structural and constraint consistent instance of c, is also a view-query consistent instance of c. $\square$

A finite set of objects OBJ is consistent if the set is closed under the *depend_on* relation, that is, for each object in the set all the objects referred by it must belong to the set, and the property of oid-uniqueness must be ensured. The following definition formalizes these concepts. Given an object o, $ref(o)$ denotes the set of identifiers in $\mathcal{OI}$ appearing in $o.v$, and, given a set of objects OBJ, $I(OBJ)$ denotes the set $\{i \mid o = (i, v, VS), o \in OBJ\}$.

Definition 13 (Consistent Set of Objects). *A (finite) set of objects OBJ is consistent iff all the following conditions hold:*

1. OID-UNIQUENESS
   $\forall o_1, o_2 \in$ OBJ, if $o_1.i = o_2.i$, then $o_1.v = o_2.v$ and $o_1.VS = o_2.VS$.
2. REFERENTIAL INTEGRITY
   $\forall o \in OBJ,\ ref(o) \subseteq I(OBJ)$. $\square$

A finite set of objects OBJ containing also view objects is closed under the *derived_by* relation if the base objects from which a view object in OBJ is derived belong to OBJ, too, as stated by the following definition.

Definition 14 (Closed Set of Objects). *A (finite) set of objects OBJ is consistent iff $\forall o \in OBJ$ such that $o.i \in \mathcal{VOI}$, $derived\_by(o.i) \subseteq I(OBJ)$.* $\square$

5.3. *Subview relationships*

An object-oriented view mechanism should keep the basic concepts of the object-oriented paradigm, so that a view can be used in any context where a class is. Following this guideline, inheritance relationships among views are supported in our model. Views are organized in a view hierarchy exactly as classes are organized in a class hierarchy. The view hierarchy is thus modeled by an ISA relationship, representing which views are subviews of other views, and must be established by the user.

The information of a view hierarchy can be expressed by two functions $VISA : \mathcal{VI} \rightarrow 2^{\mathcal{VI}}$ and $VISA^* : \mathcal{VI} \rightarrow 2^{\mathcal{VI}}$, such that given a view v, $VISA(v)$ denotes the set of direct superviews of v and $VISA^*(v)$ denotes the set of all the superviews of v, similarly to the ISA and ISA^* functions which describe the class hierarchy. An ordering $\leq_{VISA}$ on views is defined, by simply stating that $v_1 \leq_{VISA} v_2$ iff $v_2 \in VISA^*(v_1)$, exactly as the ordering $\leq_{ISA}$ is defined in [21].

However, a certain number of conditions on the well-formedness of the view inheritance hierarchy must be imposed. Those conditions concern the following aspects:

- *subtyping among the structural types*: a view must have all the attributes of its superviews; attribute domains may be specialized, the implementation for a derived attribute may be redefined and new attributes can be added;
- *behavior specialization*: a view must have all the operations of its superviews; method signatures can be redefined, by applying the covariance rule for method results and the contravariance rule for method parameters, the method implementations may be redefined and new operations may be added;
- *constraint inheritance*: on a view all the constraints of its superviews must hold; constraint redefinition is not currently supported in Chimera;
- *extent inclusion*: the extent of a view class is a subset of the extents of all its superviews.

Those conditions are formalized as follows.

Definition 15 (Int-well-defined Subview). *A subview relationship $VISA$ is int-well-defined, if, for any v_1 and v_2 such that $v_2 \leq_{VISA} v_1$, being $V_1 = (v_1, struct_1, beh_1, constr_1, state_1, mc_1, q_1)$, and $V_2 = (v_2, struct_2, beh_2, constr_2, state_2, mc_2, q_2)$, all the following conditions hold:*

- $struct_1 = \{(a'_1, T'_1, at'_1), \ldots, (a'_{k_1}, T'_{k_1}, at'_{k_1})\}$, $struct_2 = \{(a''_1, T''_1, at''_1), \ldots, (a''_{k_2}, T''_{k_2}, at''_{k_2})\}$
  and for each $i = 1 \ldots k_1$ (a''_j, T''_j, at''_j) exists, $1 \leq j \leq k_2$, such that
  - $a'_i = a''_j$
  - T''_j is a subtype of T'_i
  - $at'_i = at''_j$[16]
- $beh_1 = \{(m'_1, s'_1), \ldots, (m'_{h_1}, s'_{h_1})\}$, $beh_2 = \{(m''_1, s''_1), \ldots, (m''_{h_2}, s''_{h_2})\}$
  and for each $i = 1 \ldots h_1$ (m''_j, s''_j) exists, $1 \leq j \leq h_2$, such that
  - $m'_i = m''_j$
  - if $s''_j = T''_{j_1} \times \ldots \times T''_{j_n} \to T''_j$, then $s'_i = T'_{i_1} \times \ldots \times T'_{i_n} \to T'_i$ and T''_j is subtype of T'_i, while for each $r = 1, \ldots, n$ T'_{i_r} is a subtype of T''_{j_r}
- $constr_1 = \{(con'_1, s'_1), \ldots, (con'_{h_1}, s'_{h_1})\}$, $constr_2 = \{(con''_1, s''_1), \ldots, (con''_{h_2}, s''_{h_2})\}$
  and for each $i = 1 \ldots h_1$ (con''_j, s''_j) exists, $1 \leq j \leq h_2$, such that
  - $con'_i = con''_j$
  - $s'_i = s''_j$.

The same refinement conditions must hold for class features as well, that is, they must be satisfied by mc_2 and mc_1. □

The definition above only concerns the structure and behavior of views, and it does not consider the extensional components of views, thus no conditions are imposed on view-queries. Let us now turn to the extensional level.

Definition 16 (Ext-well-defined Subview). *A subview relationship $VISA$ is ext-well-defined, if, for any v_1 and v_2 such that $v_1 \leq_{VISA} v_2$, $[\![v_1]\!] \subseteq [\![v_2]\!]$*[17]. □

While the conditions for int-well-definedness of a hierarchy are conditions on the schema level, and thus can be checked at view definition time, the condition for ext-well-definedness is a state-dependent (time varying) condition, that can only be checked at run-time. However, since the instances of a view are computed starting from the view root classes and the view-query, we impose a number of conditions on view root classes and query. Such conditions ensure that if the ISA hierarchy on base classes is ext-well-defined, the same property holds for the VISA hierarchy upon view materialization.

The conditions we impose on view queries in subclasses are syntactic conditions, and they are quite restrictive. Actually, we should impose that the view-query of a view is subsumed by the view-queries of its superviews. However, the problem of query subsumption is undecidable in general, and, even for such query languages for which it is decidable, it has a very high complexity test [13]. For our (recursion-free) query language, query subsumption is decidable, though intractable because of negation and disjunction [13].

We thus impose the syntactical restriction that the view-query of the subview is stronger than the view-query of the superview, as formalized by the following definition. This syntactical condition ensures that the view query of the subview is subsumed by the view-query of the superview. The condition requires that: (i) the root classes of the subview are the same or subclasses of the root classes of the superview; (ii) for each rule r_1 in the view query of the subview there must be a corresponding rule r_2 in the view-query of the superview such that the body of r_1 can be obtained from the body of r_2 by adding some atoms and by replacing some class formulas with class formulas on subclasses.

Definition 17 (View-query Strengthening). *A subview relationship $VISA$ is view-query strengthening, if, for any v_1 and v_2 such that $v_1 \leq_{VISA} v_2$, being $V_1 = (v_1, struct_1, beh_1, constr_1, state_1, mc_1, q_1)$, and $V_2 = (v_2, struct_2, beh_2, constr_2, state_2, mc_2, q_2)$, both the following conditions hold:*

- if $defined\_on(v_1) = \{c_1, \ldots, c_n\}$, then $defined\_on(v_2) = \{c'_1, \ldots, c'_n\}$ and $\forall i, 1 \leq i \leq n$, either $c_i, c'_i \in \mathcal{CI}$ and $c_i \leq_{ISA} c'_i$ or $c_i, c'_i \in \mathcal{VI}$ and $c_i \leq_{VISA} c'_i$;
- if q_2 consists of the rules $H'_1 \leftarrow B'_1 \ldots H'_n \leftarrow B'_n$, then q_1 consists the rules
  $H_1 \leftarrow B_1 \ldots H_m \leftarrow B_m$, and
  - $m \geq n$,
  - $\forall j, 1 \leq j \leq m$, $\exists i, 1 \leq i \leq n$, such that
    - $B_j = c^1_j(X_1), \ldots, c^p_j(X_p), B^*_j$,
    - $B'_i = c^1_i(X_1), \ldots, c^p_i(X_p), B'^*_i$,
    - $B^*_j = B'^*_i, \tilde{B}$ where $\tilde{B}$ is any conjunction of atomic formulas,
    - $\forall k, 1 \leq k \leq p$, either $c^k_j, c^k_i \in \mathcal{CI}$ and $c^k_j \leq_{ISA} c^k_i$ or $c^k_j, c^k_i \in \mathcal{VI}$ and $c^k_j \leq_{VISA} c^k_i$. □

If a subview relationship is view-query strengthening and the corresponding subclass relationship is ext-well-defined, then the subview relationship is ext-well-defined.

We remark that, according to the definition of view-query strengthening, if a subview relationship $VISA$ is view-query strengthening, then, for any v_1 and v_2 such that $v_1 \leq_{VISA} v_2$, $new\_oid(v_1) = new\_oid(v_2)$.

Example 9. *Consider the view* `Magazine` *defined in Example 1 and the view* `VPublication` *defined as identity view of the class* `Publication`*, both belonging to the* `Bibliography` *schema of Figure 2. Let $VC_1.id =$* `Magazine`*, $VC_2.id =$* `VPublication`*,*

$VC_1.struct = \{$ (title, string, ext), (year, integer, ext), (editorial, string, ext), (month, string, ext) (manager, string, ext) $\}$.
$VC_2.struct = \{$ (title, string, ext), (year, integer, ext), (editorial, string, ext) $\}$.

The int-well-definedness of the $VISA$ relationship stating that `Magazine` $\leq_{VISA}$ `VPublication` holds, because:

- the condition on the *struct* components is verified, since $VC_2.struct \subseteq VC_1.struct$;
- the condition on the *beh* components follows from the int-well-definedness of the ISA relationship stating that `Journal` $\leq_{ISA}$ `Publication`, since $VC_2.beh = C_2.beh$, being $C_2.id =$ `Publication` and $VC_1.beh = C_1.beh \cup \{changeManager : string \rightarrow Magazine\}$, being $C_1.id =$ `Journal`;
- the condition on the *constr* components immediately follows from the int-well-definedness of the ISA relationship stating that `Journal` $\leq_{ISA}$ `Publication`, since $VC_2.constr = C_2.constr$, being $C_2.id =$ `Publication` and $VC_1.constr = C_1.constr$, being $C_1.id =$ `Journal`.

The ext-well-definedness of the $VISA$ relationship stating that `Magazine` $\leq_{VISA}$ `VPublication` follows from its view-query strengthening, since:

- $defined\_on($`Magazine`$) = \{$`Journal`$\}$,
  $defined\_on($`VPublication`$) = \{$`Publication`$\}$
  and `Journal` $\leq_{ISA}$ `Publication`;
- the view-query of `VPublication`, since `VPublication` is an identity view, is

  `VPublication(X)` $\leftarrow$ `Publication(X)`

  the view-query of `Magazine` is

  `Magazine(X)` $\leftarrow$ `Journal(X)`,
  `X.periodicity` $=$ `'weekly'`

  and `Journal` $\leq_{ISA}$ `Publication`.

◇

The generation of view object identifiers must be carefully handled when views are related by inheritance hierarchies. Recall that for a view v such that $new\_oid(v) = true$, new object identifiers are generated for each object satisfying the view-query of v. The generation of these new oids is exactly like the generation of base object identifiers upon object creation. Thus, when views are related by a subview relationship, v is a view such that $new\_oid(v) = true$ and $VISA(v) = \emptyset$, the new oids for v are incrementally generated. For the views v' such that $new\_oid(v') = true$ and $VISA(v') = \{v_1, \ldots, v_n\}, n > 0$, the objects instances of the view are by contrast extracted from the extents of the superviews, rather than being generated upon view materialization. We thus ensure that if the view-query is stronger than the view-queries of its superviews, the extent of the view is a subset of the extents of its superviews. Note that, if we had generated new oids for each view v such as $new\_oid(v) = true$, without taking into account the VISA relationships among views, the view-query strengthening of the VISA relationship would not have ensured the ext-well-definedness of the hierarchy. Thus, for object-generating views, new oids are generated upon view materialization for the views that are roots of the VISA hierarchy (which are materialized first), while for views having at least a superview the extent is determined by extracting from the superview extents those objects meeting the condition in the subview query. This process is sound though in Chimera multiple inheritance is supported, since the constraint is imposed that for multiple inheritance a common ancestor must exist.

View identifiers can be used as types for Chimera expressions. Thus, the notion of Chimera type proposed in [21] is modified to include also view class identifiers. The set of Chimera types $\mathcal{T}$ then consists of the value types in $\mathcal{VT}$ and the object types in $\mathcal{OT}$, that is, of class and view identifiers. The set of Chimera object types (that is, of types whose values are used to identify objects) is thus defined as the union of class and view identifiers, that is, $\mathcal{OT} = \mathcal{CVI}$.

The notion of type extension is easily extended to types that are view identifiers (that is, for $v \in \mathcal{VI}$) since views, like classes, have an explicit extent, thus $[\![v]\!] = V.struct.extent$, where $V.id = v$. The subtype relationship $\leq_T$ can be simply adapted to a set $\mathcal{T}$ of types containing also view identifiers in $\mathcal{VI}$, by stating that:

- if $v_1, v_2 \in \mathcal{VI}$ and $v_1 \in VISA^*(v_2)$ then $v_2 \leq_T v_1$ that is, v_2 is a subtype of v_1 (as stated for object types in $\mathcal{CI}$ [21]);
- view class identifiers in $\mathcal{VI}$ are not related by the subtype relationship with other types.

Since we consider only int-well-defined subview relationships, this is a sound definition of subtyping.

6. Schema views and database views

In this section, we "put things together", discussing the notions of schema views and database views. Moreover, we address some issues related to the use of views.

6.1. Schema views

The definition of schema view or subschema often corresponds to the concept of external schema given in the ANSI three-level schema architecture. In our approach, schema views are also intended to encapsulate base schema evolutions, as a mean to prevent that a schema update affects the base schema. Thus schema views can be useful to define external schemas as well as to create new schema versions.

The notion of schema view is similar to that of base schema, except that it consists of views instead of classes. A schema view consists of a collection of views connected by *aggregation* and *view inheritance* relationships. All views in a schema view are derived from the same schema. The schema, from which a schema view is derived, is called *root schema*. A root schema can be either a base schema or another schema view. Frequently, the definition of a schema view requires to include some base classes. We propose the notion of *identity view* to satisfy this requirement, still having the schema view consisting only of views. Given a class c, its identity view is a view v having c as root class, such that c and v are equivalent, both at the extensional and at the intensional level.

The following definition formalizes the notion of schema view. Let $\mathcal{SVI}$ denote a set of schema view identifiers.

Definition 18 (Schema View). *A schema view having as root schema a schema S*[18] *is defined as a tuple*

$$(id, VT, VCl, VMCl, VISA)$$

where

- $id \in \mathcal{SVI}$ is the schema view identifier;
- $VT \subseteq \mathcal{VT} = BVT \cup NVT$, is the set of value types included in the schema view, being BVT the set of value types imported from the root schema S and NVT the set of value types defined in the schema view;
- VCl is a finite set of definitional components of views [19];
- $VMCl \subseteq \mathcal{MC}$ is a finite set of view metaclasses corresponding to the views in VCl;
- $VISA : VCl \rightarrow 2^{VCl}$ is a total function on VCl for which the following conditions hold:

  a) $VISA$ is a DAG;

  b) $VISA$ is int-well-defined;

  c) $VISA$ is view-query strengthening.

All view and view metaclass names are distinct and for each view the corresponding view metaclass must exist. □

The following example is an example of schema view.

Example 10. *Figure 2 shows a schema view named* `Bibliography`, *directly derived from the base schema. The* `Bibliography` *schema view contains the views* `VPublication`, `Magazine` *and* `VBook`. *The* `VPublication` *view has been imported as an identity view, whereas* `Magazine` *and* `VBook` *views are derived from* `Journal` *and* `Book` *classes, respectively, and they are declared as subviews of the* `Vpublication` *view. The importation of* `Publication` *as identity view allows to define a virtual hierarchy of publications.* ◇

6.1.1. Closure of schema views A schema view is a collection of views, grouped together to form a subschema, or to model a schema evolution. One is not completely free in choosing which views to include in a schema view. In particular, if a view is included in the schema view, also the domain of each attribute as well as the components of the signature of each operation of the view must belong to the schema view. Thus, including a view into the schema view may require the inclusion of other views. In the following, we formalize these notions, which are referred to as *closure* property of a schema view.

In what follows, the term *entity* refers to an attribute, c-attribute, parameter of an operation or parameter of a c-operation of a class or view. We introduce a *client_of* relationship among classes and views in $\mathcal{CVI}$. A class (or view) c_1 is said to be *client_of* a class (or view) c_2, if some entity of c_1 has as domain the class (or view) c_2. Since a user (e.g., an application program) must receive a schema view consisting of a complete and coherent set of views, it is clear that a schema view must be closed under the *client_of* relation. Given a view v included in a schema view, the closure property involves:

1. for each entity of v, whose domain is a class, the corresponding identity view must belong to the schema view and it is the new domain of the entity;
2. for each entity of v, whose domain is a view, this view must belong to the schema view.

A closed schema view contains all the views referenced directly or indirectly by the schema view definition. Since we have chosen to model a schema view as a collection of views, if the domain of an entity of v is a class of the base schema, the corresponding identity view is introduced in the schema view, thus "virtualizing" the class without modifying it.

We remark two important aspects related to our definition of schema view closure. First, since Chimera does not require the existence of a common superclass of all the classes of the system, the closure property does not involve the inclusion of all the superviews of the views belonging to the schema. Second, the closure of a schema does not require the inclusion of all the subviews of the views belonging to the schema. Thus, a schema view is closed with respect to aggregation hierarchies, while it is not closed with respect to inheritance hierarchies. Indeed, the schema closure must contain the essential views for a schema to be consistent. In some contexts, it seems reasonable that a schema view includes a view and it does not include some of its subviews (so that some of the subviews are hidden in the schema view). As a consequence of this choice, the database view (that is, the database seen through the schema view) may contain objects whose most specific classes and views do not belong to the schema view. Such objects are seen through the schema view as instances of the most specific view to which the object belongs, among the ones included in the schema view. Note that this will require a careful propagation of updates, since an attribute could result in having different domains depending on through which schema view it is accessed. Possible solutions are: to consider the attribute as a different attribute (for example, prefixing its name with the schema name) and thus allocating different storage space; to handle the propagation through triggers which specialize the value assigned to the attribute to the required domain. The most conservative solution to avoid those problems is to constrain a schema to contain also the subclasses of the classes in the schema, for those subclasses that refine the domain of some included attribute. Since those problems are related to update propagation, we do not elaborate on them further in this paper.

To formalize the closure property of a schema view, a function id_view is introduced. It applies to a schema view and returns the schema view modified by substituting each class identifier belonging to $\mathcal{CI}$ and appearing as domain of an entity in the schema, with the identifier of the corresponding identity view, which belongs to $\mathcal{VI}$. Thus, given a schema view SV, $id\_view(SV)$ denotes a corresponding schema view having only views or value types as entity domains. Furthermore, given a view v, belonging to a schema view SV, let $dom(v)$ denote the set of value types and views which are domains of the entities of v in $id\_view(SV)$. This set can be partitioned in $vdom(v)$, only containing view identifiers, obtained as $dom(v) \cap \mathcal{VI}$, and $tdom(v)$, only containing value types, obtained as $dom(v) \cap \mathcal{VT}$.

Definition 19 (Schema View Closure). *A schema view SV is said to be closed if both the following conditions are satisfied:*

1. $\forall v \in SV.VCl$, $vdom(v) \subseteq SV.VCl$
2. $\forall v \in SV.VCl$, $tdom(v) \subseteq SV.VT$. □

We remark that, given a schema view SV, it is decidable whether SV is closed.

The following example illustrates the closure property of a schema view.

Example 11. Consider again the `Bibliography` *schema view of Figure 2. Since the view* `Vbook` *contains an attribute named* `authors` *whose domain is class* `Author`*, the domain of this attribute must be replaced by the identity view* `IdAuthor` *of the* `Author` *class, and this identity view must be included in the schema. This inclusion is propagated to the* `Address` *class because there is an* `address` *attribute in the* `Author` *class whose domain is* `Address`*, so that the identity view* `IdAddress` *is also included. The closure of the schema is depicted in Figure 5. The derivation links between* `IdAuthor` *and* `Author`*, and between* `IdAddress` *and* `Address`*, are not depicted in the figure, to point out that these views are added to the schema view to obtain a closed schema view.* ◇

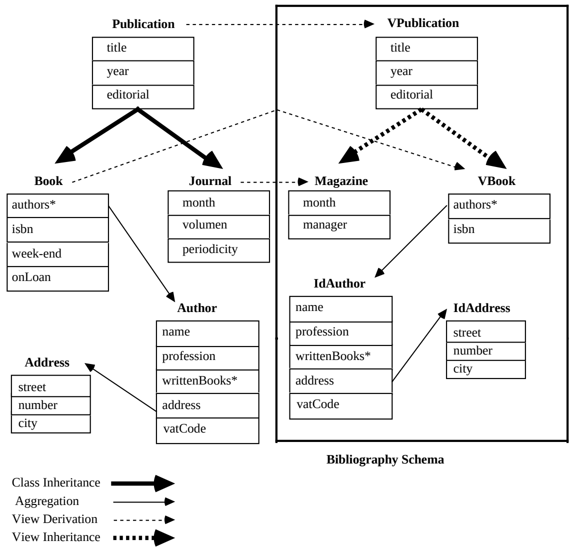

FIG. 5. Closure of the `Bibliography` view schema.

6.1.2. Global database schema Now, by using the notion of schema views, we give a formal definition of global database schema. A global database schema consists of a base schema together with a set of schema views. The schema derivation and view derivation relationships are part of the global database schema, too.

Definition 20 (Global Database Schema). *A global database schema is a tuple*

$$(bs, SVS, defined\_on^s, defined\_on),$$

where:

- bs is a base schema identifier,
- $SVS = \{sv_1, \ldots, sv_n\}, n \geq 0$, is a set of schema view identifiers,
- $defined\_on^s : \mathcal{SVI} \rightarrow \mathcal{SVI} \cup \{bs\}$ is a total function, that given a schema view returns its root schema,
- $defined\_on : \mathcal{VI} \rightarrow 2^{\mathcal{CVI}}$ is the function introduced in Subsection 5.1, that represents the view derivation relationship.

The schema must satisfy the following conditions:

1. each schema in SVS must be closed according to Definition 19;
2. for each $sv_i, sv_j \in SVS$, if $defined\_on^s(sv_j) = sv_i$, then $\forall v \in SV_j.VCl, defined\_on(v) \subseteq SV_i.VCl$, being $sv_i = SV_i.id$ and $sv_j = SV_j.id$. □

Note that the view derivation relationship may connect view and classes belonging to different schemas, thus it is part of the global database schema rather than of an individual schema view.

Example 12. *Referring to our running example, the global database schema consists of the* `Faculty-Library` *base schema and the* `Bibliography` *schema view, with the following relationships:*

- $defined\_on^s(\texttt{Bibliography}) = \texttt{FacultyLibrary}$;
- $defined\_on(\texttt{VPublication}) = \texttt{Publication}$
  $defined\_on(\texttt{Magazine}) = \texttt{Journal}$
  $defined\_on(\texttt{VBook}) = \texttt{Book}$
  $defined\_on(\texttt{IdAuthor}) = \texttt{Author}$
  $defined\_on(\texttt{IdAddress}) = \texttt{Address}$. ◇

6.2. *Object database*

In our approach, we consider a single database which is associated with the global schema, and which is shared by all the existing schemas. The database contains all the instances created from classes and views belonging to the global schema. Thus, the database consists of objects defined according to Definition 6, that can be, as a particular case, view objects as in Definition 7. An object $o = (i, v, VS)$ can be accessed from any schema including a view or class of which o is member. We remark that this view or class does not necessary belong to the set VS, since VS only contains the classes and views of which o is an instance. Suppose that an object o can be accessed through the schema view SV, the function $value$, defined in Subsection 5.2.1., is then used to provide the different aspects under which the object can be seen. Therefore, each schema view in the global schema has associated a view database. Now, we reformulate the definition of database, before giving a definition of view database.

Given a global database GS, let $\mathcal{CI}^{GS}$ denote the set of identifiers of classes belonging to the global database schema, which are those included in the base schema, that is

$$\mathcal{CI}^{GS} = \{c \mid c \in \mathcal{CI}, c = C.id, C \in S.VCl, S.id = GS.bs\}.$$

Let moreover $\mathcal{VI}^{GS}$ denote the set of identifiers of views belonging to the global database schema, which are those included in any schema view, that is

$$\mathcal{VI}^{GS} = \bigcup_{sv_i \in GS.SVS} \mathcal{VI}^i$$

where each $\mathcal{VI}^i$ is obtained as

$$\mathcal{VI}^i = \{v \mid v \in \mathcal{VI}, v = VC.id, VC \in SV_i.VCl, SV_i.id = sv_i\}.$$

Finally, $\mathcal{CVI}^{GS} = \mathcal{CI}^{GS} \cup \mathcal{VI}^{GS}$.

Definition 21 (Object Database). *Let GS be a global database schema defined according to Definition 20, a database over GS is a tuple*

$$(OT, \pi, cval)$$

where

- $OT = OB \cup OV$, is a consistent set of objects according to Definition 13, being
  - OB a set of base objects;
  - OV a set of view objects;
- π is a pair of functions

  $\pi_{OC} : \mathcal{CI}^{GS} \to 2^{\mathcal{BOI}}$, oid assignment for base classes,
  $\pi_{OV} : \mathcal{VI}^{GS} \to 2^{\mathcal{VOI}}$, oid assignment for views,

  which handle class extents;
- $cval$ is a total function $cval : \mathcal{CVI}^{GS} \to \mathcal{V}$, such that $\forall c \in \mathcal{CVI}^{GS} cval(c)$ is a legal value for $stype(C.mc)$, being $C.id = c$, that is, the function $cval$ assigns values to the class attributes of C. □

For an object database to be consistent, we require that it satisfies a number of conditions, as stated by the following definition. These conditions are mainly related to the proper assignment of objects to classes and views.

Definition 22 (Consistent Object Database). *Let GS be a global database schema defined according to Definition 20, a consistent object database over GS is a tuple*

$$(OT, \pi, cval)$$

defined according to Definition 21 such that:

(i) $\forall o \in OT, o$ is a consistent instance of each class and view in $o.VS$, that is $o.v$ holds all the values for the attributes of classes in $o.VS$, thus satisfying conditions stated in Definition 8;

(ii) the ISA component of GS is ext-well-defined;

(iii) the VISA components of GS are ext-well-defined;

(iv) $\forall c \in \mathcal{CVI}^{GS}$, $c = C.id$,
$cval(C).extent = \pi(c)$ and
$cval(C).proper\_extent = \pi(c) \setminus \bigcup_{c' \text{s.t.} c \in ISA(c')} \pi(c')$[20];

(v) $\forall o \in OT, \forall c \in \mathcal{CVI}^{GS}$, $c = C.id$, $o.i \in cval(C).proper\_extent$ iff $c \in o.VS$;

(vi) $\forall o \in OT, o.VS \subseteq \mathcal{CVI}^{GS}$ and
  - $\forall o \in OB, o.VS \subseteq \mathcal{CVI}^{GS}$ and $\exists c \in o.VS$ such that $c \in \mathcal{CI}^{GS}$, while
  - $\forall o \in OV, o.VS \subseteq \mathcal{VI}^{GS}$;

(vii) $\forall v \in \mathcal{VI}^{GS}$, if $next\_oid(v) = false$ and $defined\_on(v) = \{c_1, \ldots, c_n\}$, then $\pi(v) \subseteq \bigcup_{i=1,\ldots,n} \pi(c_i)$;

(viii) $\forall o \in OV$, $\exists v \in \mathcal{VI}^{GS}$ such that $v \in o.VS$ and $next\_oid(v) = true$;

(ix) $\forall o \in OT$, $\forall v \in o.VS \cap \mathcal{VI}^{GS}$, $oid(v) = true$;

(x) $\forall v \in \mathcal{VI}^{GS}$ such that $oid(v) = false$, $\pi(v) = \emptyset$;

(xi) $\forall c_1, c_2 \in \mathcal{CVI}^{GS}$

– if $c_1, c_2 \in \mathcal{CI}^{GS}$

$\pi(c_1) \cap \pi(c_2) \neq \emptyset \Rightarrow \exists c_3 \in \mathcal{CI}^{GS}$ such that $c_1 \leq_{ISA} c_3, c_2 \leq_{ISA} c_3$;

– if $c_1, c_2 \in \mathcal{VI}^{GS}$ and $new\_oid(c_1) = new\_oid(c_2) = true$

$\pi(c_1) \cap \pi(c_2) \neq \emptyset \Rightarrow \exists c_3 \in \mathcal{VI}^{GS}$ such that $c_1 \leq_{VISA} c_3, c_2 \leq_{VISA} c_3$;

– if $c_1, c_2 \in \mathcal{VI}^{GS}$ and $new\_oid(c_1) = new\_oid(c_2) = false$

$\pi(c_1) \cap \pi(c_2) \neq \emptyset \Rightarrow (\exists c_3 \in \mathcal{VI}^{GS}$ such that $c_1 \leq_{VISA} c_3, c_2 \leq_{VISA} c_3) \vee$
$\vee\ (\exists c_4 \in \mathcal{VI}^{GS}$ such that $c_4 \in defined\_on^*(c_1) \wedge c_4 \in defined\_on^*(c_2)))$;

– if $c_1, c_2 \in \mathcal{VI}^{GS}$ and $new\_oid(c_1) = true$ while $new\_oid(c_2) = false$

$\pi(c_1) \cap \pi(c_2) \neq \emptyset \Rightarrow c_2 \in defined\_on^*(c_1)$;

– if $c_1 \in \mathcal{VI}^{GS}$, $c_2 \in \mathcal{CI}^{GS}$

$\pi(c_1) \cap \pi(c_2) \neq \emptyset \Rightarrow c_2 \in defined\_on^*(c_1) \wedge new\_oid(c_1) = false$.

□

In the definition above, condition *(vi)* requires that each base object is instance of at least one base class whereas each view object only belongs to views. Condition *(vii)* ensures that the extent of an object-preserving view is contained in the union of the extents of its root classes (we recall that an object-preserving view could select objects from different classes), while condition *(viii)* imposes that the views to which a view object belongs are object-generating views. Finally, condition *(xi)* states the conditions under which two classes or views may have non disjoint extents: they can be either both base classes or both object-generating views with a common ancestor in the inheritance hierarchy; they can be both object-preserving views with a common ancestor in the inheritance hierarchy or a common root class; or they can finally be a base class and an object-preserving view such that the view is derived from the base class.

Definition 23 (Global Database). *Let GS be a global database schema, then a global database on GS is a tuple exactly like in Definition 21, of the form $(OT, \pi, cval)$ such that OT is closed under the view derivation relationship according to Definition 14.* □

Each schema views specifies a different view of the global database, as stated by the following definition.

Definition 24 (Database View). *Let SV be a schema view on a consistent object database ODB, then a database view is a tuple exactly like in Definition 21, of the form $(OT_{SV}, \pi_{SV}, cval_{SV})$ satisfying the following further conditions:*

(i) $I(OT_{SV}) \subseteq I(ODB.OT)$[21] is such that

– $\forall o \in OT_{SV}$, $\exists v \in \mathcal{VI}^{SV}$ such that $o.i \in \pi(c)$;

– $\forall o \in OT_{SV}$, let $o' \in ODB.OT$ be the object such that $o.i = o'.i$; then

∗ $o.v$ contains[22] $o'.v$, and

∗ $o.VS = o'.VS \cap \mathcal{CVI}^{SV} \cup ms(\{v \mid v \in \mathcal{VI}^{SV}, \exists v' \in o.VS, v' \notin \mathcal{VI}^{SV}, v' \leq_{VISA} v\}$
where, given a set of views VS, $ms(VS)$ denotes the set
$\{v \mid v \in \mathcal{VI}, \not\exists v' \in \mathcal{VI}$ such that $v' <_{VISA} v\}$[23];

(ii) π_{SV} is the restriction of the function in $ODB.\pi$ to views in $\mathcal{VI}^{SV}$;

(iii) $cval_{SV}$ is the restriction of the function in $ODB.cval$ to views in $\mathcal{VI}^{SV}$. □

6.3. Object references and contexts

Another important aspect concerning views is how object references are solved, since "different" objects can be identified by the same object identifier. Indeed, different views of the same object are allowed, depending on the *context* in which the object is considered.

In our approach, the class (or view) the referenced object must be seen an instance of, is chosen among the ones belonging to the current schema view, taking into account the context of the object reference. The context of an object reference is simply determined by the static type of the object in the expression containing the object reference. Indeed, each object reference in each Chimera expression is assigned a single static type [9]. Thus, it is possible in each expression to derive a unique context for each expression denoting an object (object reference). The context of an object reference can be derived from the types declared for the variables in the expression and from schema information.

As far as attribute access is concerned, an attribute access $e.a$ is solved by simply returning $value(o, t^s(e)).a$ where $value$ is the function defined in Section 5, $t^s(e)$ is the static type of the object reference e in the considered expression and o is the object to which reference e is instantiated. Note that, for the expression containing the reference to be a legal expression, $t^s(e)$ must be a view identifier belonging to the schema view of the user that has written the expression and that attribute a must belong to the structure of $t^s(e)$ in this schema view.

Method dispatching may however become more complicated when several method implementations are applicable to a method invocation. In general, the implementation specified in the most specific class of the invocation receiver is executed, as it is the one that *most closely matches* the invocation. However, when objects belong to classes and views not related by an inheritance relationship, the choice of the method implementation that "most closely matches" the invocation is not obvious. We have addressed the problem of dispatching for objects belonging to multiple most specific classes in [9][24]. The approach is based on the idea that each object has in each context a *preferred* class, among its most specific ones. This approach can be easily adapted to our framework. Each method invocation is dispatched choosing the implementation in the preferred class in the current context. This approach supports a context-dependent behavior, as the same method invocation may be dispatched differently, and thus may return different results and perform different updates, depending on the context where the method is invoked. This approach is based on a total ordering on views, defined consistently with the inheritance ordering, in such a way that a method invocation is dispatched by executing:

- the method in the view which is the static type of the object reference, if this view is among the most specific ones to which the object belongs;
- the minimum with respect to the considered total ordering, of a set of views that verify the following conditions:
  - they are subviews of the static type of the object reference (this ensures that no run-time type errors occur),
  - they belong to the current schema view,

  otherwise;
- the method in the root class of the view which is the static type of the object reference, if the two cases above are not able to dispatch the method (resolution in the root schema).

Note that while the static type of the object reference certainly belongs to the user schema (otherwise the expression containing the reference would not be correct), this may not be true for any subview/subclass of this type. Thus, we remark that, though we store a single object database, the objects stored in the database behave differently depending on through which schema view they are accessed. The behavior does not only depend from the static type of the object reference, but also from the schema view in which the object reference is contained.

7. Conclusions and future work

In this paper we propose a formal model of views for object-oriented databases. The proposed view mechanism is as powerful as existing view mechanisms and can be easily adapted to any object-oriented data model. An important aspect of our view mechanism is that different views of a single object database are provided, through different schema views. A schema view is a coherent set of views. The schema view through which an object database is accessed also influences the object behavior, thus providing a context-dependent behavior. Schema views also support a mechanism of schema versions, such that a single database is shared by all the schemas.

The model we propose in this paper is the first, as far as we know, formal model for views in object-oriented databases. It can be used as a starting point for investigating several interesting issues related to object-oriented views. As an example, update propagation is being investigated on this model. The view update problem has been widely investigated in the relational context. As noted in [40], the existence of object identifiers makes easier updating object-oriented views than relational views, because it is possible to establish a mapping among a view instance and its base object(s). In analyzing object-oriented view updates, it is necessary to distinguish between updating object-preserving views and object-generating views. To our knowledge, all the proposed approaches [4, 40] have only considered object-preserving views. View updates are quite straightforward when a view model only includes object-preserving views, because they are automatically propagated to the base objects: both the view instance and its base object have the same object identifier. The model described in [33] also uses an algebra with object-preserving operators, but view updates are more complex because view instances can have additional storage and the data model does not support multiple class instantiation. Since Chimera supports multiple class instantiation, the approach to view update presented in [40] is applicable to the object-preserving views considered in our view model. In the case of object-generating views, the DERIVED BY table can be employed for holding the correspondence between each generated view instance and the base objects from which it is derived. Through this table, the update operations (e.g. insert, delete, modify) can be propagated to the base objects. We believe that the framework presented in [27] for view updates is appropriate to set up our proposal for updating object-generating views in Chimera. Moreover, object-oriented data models offer the possibility of specifying in methods how to propagate ambiguous updates, such as deletions on views defined as joins, which are forbidden in the relational context.

A topic which is strictly related to update propagation concerns integrity constraints. The presence of integrity constraints introduces new issues in the design of a view mechanism. Indeed, a view definition is affected by the constraints of its root classes. If it seems coherent that a view modifies (hides, adds or redefines) behavior or structure of the base classes from which is derived, it is not so obvious which modifications are possible on constraints. It is clear that hiding constraints should be allowed in a view mechanism supporting schema evolution, because, for example, hiding an instance attribute implies hiding the constraints that use this attribute. However, the ability to hide constraints can raise problems in update propagation. Problems may arise if one creates an instance of a view not satisfying a constraint which holds on the root class but which is hidden in the view. Since the instance cannot be inserted in the root class, the update cannot be propagated.

Another interesting topic of future work concerns approaches for view materialization in Chimera. In most view models the extension of a view is not stored, but rather view objects are derived from the view query upon demand. However, materialization approaches for object-oriented views have been recently proposed. In [11], the model described in [1] is used to simulate schema changes. A materialization approach for the MultiView model [33] is presented in [28], providing the necessary update operations to enforce the consistency of the materialized views. If multiple class instantiation is supported, as in the case of Chimera, the materialization of object-preserving views has important advantages with respect to the relational views. The storage overhead decreases because the materialization does not involve storage duplication, but only marking that the base objects satisfying the view query are also instances of the view (a reference to the view identifier can be added in the object). Of course, if a view has additional attributes it is necessary to allocate storage for them. The cost of maintaining the view instances consistent upon changes to base objects also decreases. By contrast, materializing object-generating views presents problems similar to those of relational view materialization.

The use of triggers for handling views is an interesting possibility, as suggested in [18] for the relational context. If views are materialized (that is, stored in the database) rules can monitor dynamic changes to base data and modify relevant views. If, by contrast, virtual views are supported, rules can dynamically detect queries on virtual views and transform them into queries on base data, by composing the user query with the query defining the view. Finally, rules can propagate view updates to base data.

As far as schema evolution is concerned, in our opinion there are two fundamental approaches to support schema evolution: modifying base classes, or defining a view that realizes the update (thus, simulating it). Obviously, each approach has some advantages over the other; thus, we think that both should be supported and the user should be free to choose the most adequate for the schema update to be performed. If the first approach is taken, modifications must be propagated from the modified class to the views derived from it; we are currently investigating how this propagation can be performed. Concerning the second approach, we are investigating how the taxonomy of object-oriented schema changes proposed in [6, 44] can be extended to Chimera, and how the proposed view mechanism can be exploited to support all the possible changes. Finally, other interesting topics of future work include the extension of the model by taking into account all the Chimera capabilities, that is, logical integrity constraints and triggers. Recursive view definitions may also be considered.

Acknowledgments

We wish to thank Günter Kniesel and Thomas Lemke for useful comments both on the presentation and on the technical contents of a preliminary version of this paper. Thanks are also due to the anonymous referees for their helpful comments.

Notes

1. We relate and compare in detail the features of our model with features of other view models in Section 3.
2. A Chimera is a monster of the Greek mythology with a lion's head, a goat's body and a serpent's tail; each of them represents one of the three components of the language.
3. Actually, in Chimera the operation implementation may be defined in an external programming language, but we do not consider this case because it heavily depends on the external language which is used. Thus, we consider here only implementations expressed in the Chimera language itself.
4. We do not allow oids to be explicitly manipulated by the user, thus oids cannot appear in Chimera formulas.
5. Only side-effect free methods can be employed in queries.
6. Chimera formulas also include event and constraint formulas, but those kinds of formulas are not relevant for query and view definitions.
7. $=$ denotes identity, $==$ shallow value equality and $==_d$ deep value equality.
8. Class formulas cannot be negated.
9. An exception is represented by [1] but, as we will see, that model does not support truly object-generating views in that a query returns a set of tuples, that are converted to new objects outside the query language.
10. As we will see in Section 6, given a class c, its identity view is a view v having c as root class, such that c and v are equivalent, both at the extensional and at the intensional level.
11. Any attribute expressed with a path expression can be as well declared as derived and its implementation specified by Chimera deductive rules. Path expressions are therefore a shorthand of deductive rules specifying a navigation through an aggregation hierarchy. For example the declaration `city: Author.address.city` could be expressed by the declaration `city: string derived`, being its implementation expressed by the following deductive rule:

```
self.city = X ← Author(Y), Y.address.city = X
                derived-by(self, Author, Y)
```

12. We refer with the term root classes to the views and classes a certain view has been defined on. Thus, we sometimes improperly denote as classes a set of classes and views. Whenever confusion may arise, we adopt the term base classes to distinguish classes from views.
13. The notion of source of an attribute for base classes only is formally defined in [9].
14. We say that a record value $record\text{-}of(a_1 : v_1, \ldots, a_n : v_n)$ *contains* a record value $record\text{-}of(a'_1 : v'_1, \ldots, a'_m : v'_m)$ if $m \leq n$ and, $\exists f : \{1, \ldots, m\} \rightarrow \{1, \ldots, n\}$ total injective function, such that $\forall i, 1 \leq i \leq n$, $a'_i = a_{f(i)}$ and $v'_i = v_{f(i)}$.
15. We recall that Chimera constraints are expressed in denial form, that is they specify the inconsistent states as in [17].
16. The type of an attribute may not be changed from extensional to derived nor vice-versa.
17. Equivalently, if $V_1.state.extent \subseteq V_2.state.extent$, being $V_1.id = v_1$ and $V_2.id = v_2$.
18. S may be either a base schema or a schema view.
19. The *definitional component* of a view $VC = (id, struct, beh, constr, state, mc, q)$ is $(id, struct, beh, constr, mc, q)$, that is, its state-independent components.
20. For views the $VISA$ function must be considered instead of the ISA function.
21. Recall that given a set of objects OBJ, $I(OBJ)$ denotes the set of identifiers of objects in OBJ.
22. The notion of containment between record value types has been specified in Subsection 5.2.2..
23. $<_{VISA}$ denotes the non-reflexive relation obtained from the order $\leq_{VISA}$ as follows: $v_1 <_{VISA} v_2$ if and only if $v_1 \leq_{VISA} v_2$ and $v_1 \neq v_2$.
24. Actually, in [9] we have considered only base classes, but considering views does not introduce new problems with respect to dispatching.

References

1. Abiteboul,S. and Bonner,A. (1991). Objects and Views. In J. Clifford and R. King, editors, *Proc. of the ACM SIGMOD Int'l Conf. on Management of Data*, pages 238–247.
2. A.D.B. S.A, Paris. *Matisse 2.3 Tutorial*, 1995.
3. Albano, A., Bergamini, R., Ghelli, G., and Orsini, R. (1993). An Object Data Model with Roles. In R. Agrawal, S. Baker, and D. Bell, editors, *Proc. Nineteenth Int'l Conf. on Very Large Data Bases*, pages 39–51.
4. Amer-Yahia, S., Breche, P., and Souza dos Santos, C. (1996). Object Views and Updates. In *Proc. of Journées Bases de Données Avancées - BDA'96*.
5. Atkinson, M., Bancilhon, F., DeWitt, D., Dittrich, K., Maier, D., and Zdonik, S. (1989). The Object-Oriented Database System Manifesto. In W. Kim et al., editors, *Proc. First Int'l Conf. on Deductive and Object-Oriented Databases*, pages 40–57, 1989.
6. Banerjee, J., Kim, W., Kim, H.K., and Korth, H. (1987). Semantics and Implementation of Schema Evolution in Object-Oriented Databases. In *Proc. of the ACM SIGMOD Int'l Conf. on Management of Data*.
7. Beeri, C. (1989). Formal Models for Object Oriented Databases. In W. Kim et al., editors, *Proc. First Int'l Conf. on Deductive and Object-Oriented Databases*, pages 370–395.
8. Bertino, E. (1992). A View Mechanism for Object-Oriented Databases. In M. L. Brodie and S. Ceri, editors, *Proc. Third Int'l Conf. on Extending Database Technology*, pages 136–151.
9. Bertino, E. and Guerrini, G. (1995). Objects with Multiple Most Specific Classes. In W. Olthoff, editor, *Proc. Ninth European Conference on Object-Oriented Programming*, number 952 in Lecture Notes in Computer Science, pages 102–126, Aarhus (Denmark).
10. Bertino, E. and Martino, L.D. (1993). *Object-Oriented Database Systems - Concepts and Architecture*. Addison-Wesley.
11. Breche, P., Ferrandina, F., and Kuklok, M. (1995). Simulation of Schema Changes using Views. In N. Revell and A M. Tjoa, editors, *Proc. Sixth International Conference and Workshop on Database and Expert Systems Applications*, number 978 in Lecture Notes in Computer Science, pages 247–258.
12. Breitl, R., Maier, D., Otis, A., Penney, J., Schuchardt, B., Stein, J., Williams, E.H., and Williams, M. (1989). The GemStone Data Management System. In W. Kim and F. H. Lochovsky, editors, *Object-Oriented Concepts, Databasases, and Applications*, pages 283–308. Addison-Wesley.
13. Buchheit, M., Jeusfeld, M., Nutt, W., and Staudt, M. (1994). Subsumption Between Queries to Object-Oriented Databases. In M. Jarke, J. Bubenko, and K. Jeffery, editors, *Proc. Fourth Int'l Conf. on Extending Database Technology*, number 779 in Lecture Notes in Computer Science, pages 15–22. Extended version in *Information Systems*, 19(1).
14. Cardelli, L. (1988). Types for Data Oriented Languages. In J. W. Schmidt, S. Ceri, and M. Missikoff, editors, *Proc. First Int'l Conf. on Extending Database Technology*, Lecture Notes in Computer Science, pages 1–15.
15. Cattel, R. (1996). *The Object Database Standard: ODMG-93*. Morgan-Kaufmann.
16. Ceri, S., Gottlob, G., and Tanca, L. (1990). *Logic Programming and Databases*. Springer-Verlag, Berlin, 1990.
17. Ceri, S. and Widom, J. (1990). Deriving Production Rules for Constraint Maintenance. In *Proc. Sixteenth Int'l Conf. on Very Large Data Bases*, pages 566–577, Brisbane, Australia.
18. Ceri, S. and Widom, J. (1991). Deriving Production Rules for Incremental View Maintenance. In G. M. Lohman, A. Sernadas, and R. Camps, editors, *Proc. Seventeenth Int'l Conf. on Very Large Data Bases*, Barcelona, Spain.
19. Deux, O. et al (1990). The Story of 0_2. *IEEE Transactions on Knowledge and Data Engineering*, 2(1):91–108.
20. Gottlob, G., Schrefl, M., and Rock, B. (1994). Extending Object-Oriented Systems with Roles. *ACM Transactions on Information Systems*. To appear.
21. Guerrini, G., Bertino, E., and Bal, R. (1994). A Formal Definition of the Chimera Object-Oriented Data Model. Technical Report IDEA.DE.2P.011.01, ESPRIT Project 6333. Submitted for publication.
22. Ishikawa, H., Izumida, Y., Kawato, N., and Hayashi, T. (1992). An Object-Oriented Database System and its View Mechanism for Schema Integration. In *Proc. of the Second Far-East Workshop on Future Databases*, pages 194–200.
23. Kifer, M., Kim, W., and Sagiv, Y. (1992). Querying Object-Oriented Databases. In M. Stonebraker, editor, *Proc. of the ACM SIGMOD Int'l Conf. on Management of Data*, pages 393–402.
24. Kim, W. and Chou, T. (1988). Versions of Schema for Object-Oriented Databases. In *Proc. Fourteenth Int'l Conf. on Very Large Data Bases*, pages 148–159.
25. Kim, W. (1993). Object-Oriented Databases Systems: Promises, Reality and Future. In R. Agrawal, S. Baker, and D. Bell, editors, *Proc. Nineteenth Int'l Conf. on Very Large Data Bases*, pages 676–687.
26. Kim, W. (1994). UniSQL/X Unified Relational and Object-Oriented Database System. In *Proc. of the ACM SIGMOD Int'l Conf. on Management of Data*, page 481.
27. Kim, W. and Kelley, W. (1995). On View Support in Object-Oriented Database Systems. In W. Kim, editor, *Modern Database System*, pages 108–129. ACM Press.

28. Kuno, H.A. and Rundensteiner, E.A. (1995). Materialized Object-Oriented Views in MultiView. In *Proc. Fifth International IEEE Workshop on Research Issues in Data Engineering - Distributed Object Management.*

29. Kuno, H.A. and Rundensteiner, E.A. (1996). The MultiView OODB View System: Design and Implementation. *Theory and Practice of Object Systems*, 2(3):203–225.

30. Motschnig-Pitrik, R. (1996). Requirements and Comparison of View Mechanisms for Object-Oriented Databases. *Information Systems*, 21(3):229–252.

31. Ohori, A. and Tajima, K. (1994). A Polimorphic Calculus for Views and Object Sharing. In *Proc. of the Thirteenth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, pages 255–266.

32. Penney, D.J. and Stein, J. (1987). Class Modification in the GemStone Object-Oriented DBMS. In *Proc. Second Int'l Conf. on Object-Oriented Programming: Systems, Languages, and Applications.*

33. Ra, Y.G., Kuno, H.A., and Rundensteiner, E.A. (1994). A Flexible Object-Oriented Database Model and Implementation for Capacity-Augmenting Views. Technical report, Department of Electronic Engineering and Computer Science, University of Michigan.

34. Ra, Y. and Rundensteiner, E.A. (1995). A Transparent Schema Evolution System Based on Object-Oriented View Technology. In *Proc. Eleventh IEEE Int'l Conf. on Data Engineering*, pages 165–172.

35. Richardson, J. and Schwarz, P. (1991). Aspects: Extending Objects to Support Multiple, Independent Roles. In J. Clifford and R. King, editors, *Proc. of the ACM SIGMOD Int'l Conf. on Management of Data*, pages 298–307.

36. Rundensteiner, E.A. (1992). A Methodology for Supporting Multiples Views in Object-Oriented Databases. In *Proc. Eighteenth Int'l Conf. on Very Large Data Bases*, pages 187–198.

37. Rundensteiner, E.A. (1995). The MultiView Approach to the Object-Oriented View Classification Problem. Technical report, Department of Electronic Engineering and Computer Science, University of Michigan.

38. Schilling, J. and Sweeney, P. (1989). Three Steps to Views: Extending the Object-Oriented Paradigm. In *Proc. Fourth Int'l Conf. on Object-Oriented Programming: Systems, Languages, and Applications*, pages 353–361.

39. Scholl, M., Laasch, C., and Tresch, M. (1990). Views in Object-Oriented Databases. In *Proc. Second International Workshop on Foundations of Models and Languages for Data and Objects*, pages 37–58.

40. Scholl, M., Laasch, C., and Tresch, M. (1991). Updatable Views in Object-Oriented Databases. In M. Kifer et al., editors, *Proc. Second Int'l Conf. on Deductive and Object-Oriented Databases*, number 566 in Lecture Notes in Computer Science, pages 189–207.

41. Souza dos Santos, C. (1995). Design and Implementation of Object-Oriented Views. In N. Revell and A M. Tjoa, editors, *Proc. Sixth International Conference and Workshop on Database and Expert Systems Applications*, number 978 in Lecture Notes in Computer Science, pages 91–102.

42. Souza dos Santos, C., Abiteboul, S., and Delobel, C. (1994). Virtual Schemas and Bases. In M. Jarke, J. Bubenko, and K. Jeffery, editors, *Proc. Fourth Int'l Conf. on Extending Database Technology*, number 779 in Lecture Notes in Computer Science, pages 81–94.

43. Staudt, M., Jarke, M., Jeusfeld, M., and Nissen, H. (1993). Query Classes. In S. Tsur, S. Ceri, and K. Tanaka, editors, *Proc. Third Int'l Conf. on Deductive and Object-Oriented Databases*, number 760 in Lecture Notes in Computer Science, pages 283–295.

44. Zicari, R. (1992). A Framework for Schema Udates in an Object-Oriented Database System. In F. Bancilhon, C. Delobel, and P. Kanellakis, editors, *Building an Object-Oriented Database System: The Story of O_2*, pages 146–182. Morgan-Kaufmann.

Appendix

1.1. Formats for importing features in view specification

In the `IMPORTED-FEATURES` clause of the view definition statement, `ListOfImpOper`, `ListOfImpConst`, `ListOfImpCattrib`, `ListOfCoper` and `ListOfImpCconst` denote the lists of features imported from root classes. For each feature, the associated list contains one or more items whose format can be one of the following:

a) `pName [of className]`, indicating that the feature named `pName` is imported from the root class named `className`. The specification of the class name is optional, except if the view is derived from several root classes having a feature of the same kind named `pname`.

b) `- pname [of className]`, indicating that the feature named `pname` is hidden[1]. As in the previous case, it is mandatory to specify the class name if name conflicts arise.

c) `all [of ListofClassNames]`, indicating that the view imports all the features of the class (or classes) specified. If no class is specified, the view imports all the features of all its root classes.

d) `name1 [of className] as name2`, indicating that the feature named `name1` is renamed as `name2` in the view. As in the first case, it is mandatory to specify the class name if name conflicts arise.

e) `attName: typeName derived`, indicating that a view attribute named `attName` is derived, with domain type `typeName`. Its implementation must be given in the view implementation.

f) `attName : className.`$\mathtt{a_1} . \cdots . \mathtt{a_n}$, specifying that the view attribute `attName` corresponds to the nested attribute $\mathtt{a_1} . \cdots . \mathtt{a_n}$ of the class `className`. `className` is one of the root classes of the considered view. The expression `className.`$\mathtt{a_1} . \cdots . \mathtt{a_n}$ is very similar to commonly used path expressions[2]. Thus, for $1 \leq i \leq n-1$, $\mathtt{a_i}$ must be an object valued attribute, while $\mathtt{a_n}$ can be either an object valued or a value attribute.

Formats e) and f) are allowed only for attributes.

Table A1 shows the correspondence between those formats and the corresponding items in the view signature.

TABLE A1. Correspondence between clauses of the view definition statement and components of view signature

1. for each item $\mathtt{a_i ofclass c_i}$ in `ListOfImpAttrib` the item (a_i, T_i, at_i) is added to $iStruct(V).inst$ if the item $(a_i, T_i, at_i) \in C_i.struct$, being $C_i.id = c_i$ (that is, if C_i is the class identified by c_i);
2. for each item $\mathtt{a_i}$ in `ListOfImpAttrib` the item (a_i, T_i, at_i) is added to $iStruct(V).inst$ if the item $(a_i, T_i, at_i) \in C_i.struct$, being C_i the only[(1)] class among the ones identified by an element of the `RootClasses` list in the `FROM` clause of V, containing in its $struct$ component an item whose first component is a_i;
3. for each item $\mathtt{a_i ofclass c_i as a_j}$ in `ListOfImpAttrib` the item (a_j, T_i, at_i) is added to $iStruct(V).inst$ if the item $(a_i, T_i, at_i) \in C_i.struct$, being $C_i.id = c_i$;
4. for each item $\mathtt{a_i as a_j}$ in `ListOfImpAttrib` the item (a_j, T_i, at_i) is added to $iStruct(V).inst$ if the item $(a_i, T_i, at_i) \in C_i.struct$, being C_i as in (2);
5. if `ListOfImpAttrib` contains the keyword `all`, then $iStruct(V).inst = \bigcup_{i=1,\ldots,n} C_i.struct$, being $\{C_1, \ldots, C_n\}$ the classes identified by the elements of the `RootClasses` list in the `FROM` clause of V;
6. if `ListOfImpAttrib` contains the keyword `all of` $\mathtt{c_1, \ldots, c_n}$, then $iStruct(V).inst = \bigcup_{i=1,\ldots,n} C_i.struct$, being $C_i.id = c_i$, for each $i, 1 \leq i \leq n$;
7. for each item $-\mathtt{a_i ofclass c_i}$ in `ListOfImpAttrib` the item (a_i, T_i, at_i) is removed from $iStruct(V).inst$ if the item $(a_i, T_i, at_i) \in C_i.struct$, being $C_i.id = c_i$[(2)];
8. for each item $-\mathtt{a_i}$ in `ListOfImpAttrib` the item (a_i, T_i, at_i) is added to $iStruct(V).inst$ if the item $(a_i, T_i, at_i) \in C_i.struct$, being C_i as in (2);
9. for each item $\mathtt{a_i = c_i.a_1. \cdots .a_n}$ in `ListOfImpAttrib` the item $(a_i, T_i, derived)$ is added to $iStruct(V).inst$ if T_i is the type of $c_i.a_1. \cdots .a_n$[(3)];
10. for each item $\mathtt{a_i : T_i derived}$ in `ListOfImpAttrib` the item $(a_i, T_i, derived)$ is added to $iStruct(V).inst$[(4)].

(1) If more than one root class of V contains an attribute whose name is a_i, then the specification of the class from which the attribute must be taken is mandatory.

(2) We assume that the keyword `all` implicitly added to the clause has been already considered by applying items 5 and 6.

(3) The type of $c_i.a_1 \cdots .a_n$ is obtained by making use of the following rule

$$\frac{e : cc \in \mathcal{OT} (a, T, at) \in C.struct c = C.id}{e.a : T}$$

starting from the base

$$\frac{}{c : c} \quad c \in \mathcal{OT}.$$

(4) In the case of derived attributes (for example, items (9) and (10)), the deductive rules specifying the attribute implementation must be present in the view implementation.