Materialization and Its Metaclass Implementation

Mohamed Dahchour, Alain Pirotte, *Member*, *IEEE Computer Society*, and Esteban Zimányi

Abstract—Materialization is a powerful and ubiquitous abstraction pattern for conceptual modeling that relates a class of categories (e.g., models of cars) and a class of more concrete objects (e.g., individual cars). This paper presents materialization as a generic relationship between two classes of objects and describes an abstract implementation of it. The presentation is abstract in that it is not targeted at a specific object system. The target system is supposed to provide: 1) basic object-modeling facilities, supplemented with an explicit metaclass concept and 2) operations for dynamic schema evolution like creation or deletion of a subclass of a given class and modification of the type of an attribute of a class. The presentation is generic in that the semantics of materialization is implemented in a metaclass, which is a template to be instantiated in applications. Application classes are created as instances of the metaclass and they are thereby endowed with structure and behavior consistent with the generic semantics of materialization.

Index Terms—Conceptual modeling, generic relationship, object orientation, metaclass, materialization, inheritance.

1 INTRODUCTION

Conceptual modeling is the activity of formalizing some aspects of physical and social systems for purposes of understanding and communication. Conceptual models are typically built in the early stages of system development, preceding design and implementation. But, conceptual models can also be useful even if no system is contemplated: They then serve to clarify ideas about structure and functions in a perception of a part of the world.

Advances in conceptual modeling involve narrowing the gap between real-world concepts and their representation in conceptual models by identifying powerful abstractions which allow for an accurate and intuitive representation of application domains [1], [2], [3]. Thus, more powerful conceptual models help improve the mastering of the software-development process and the quality of the final applications.

Generic relationships in object and semantic models are such powerful abstraction mechanisms. They are high-level templates for relating classes of objects. Well-known generic relationships include generalization, classification, and aggregation. Recent research on conceptual modeling has studied other generic relationships like *materialization* [4], *ownership* [5], *role* [6], *grouping* [7], *viewpoint* [8], [9], *versioning* [10], *reference* [11], *realization* [12], and *generation* [13]. These generic relationships naturally model phenomena typical of complex application domains whose semantics escapes direct representation with classical relationships. A review of generic relationships can be found in [2], [3].

- M. Dahchour and A. Pirotte are with the Université catholique de Louvain, IAG School of Management, Information Systems Unit, 1 Place des Doyens, B-1348 Louvain-la-Neuve, Belgium. E-mail: dahchour@isys.ucl.ac.be, pirotte@info.ucl.ac.be.
- E. Zimányi is with the Université Libre de Bruxelles, Faculty of Engineering, CP 165/15, 50 Av. F. Roosevelt, B-1050 Brussels, Belgium. E-mail: ezimanyi@ulb.ac.be.

Manuscript received 16 Sept. 1996; accepted 22 May 2000.
For information on obtaining reprints of this article, please send e-mail to: tkde@computer.org, and reference IEEECS Log Number 104315.

Languages for conceptual modeling can substantially ease the task of modelers if they are enriched with a variety of generic relationships. This paper deals with one such extension called materialization. It is a powerful and ubiquitous semantic pattern that relates a class of abstract categories (e.g., models of cars) and a class of more concrete objects (e.g., individual cars). Its semantics is defined in terms of the usual *isA* (generalization) and *isOf* (classification) abstractions, and of a class/metaclass correspondence. New and powerful attribute-propagation (i.e., inheritance) mechanisms are naturally associated with materialization.

Like other classical abstractions, materialization is a generic relationship, that is, a template to be instantiated in applications. Application classes can thus be provided with structure and behavior consistent with the semantics of materialization. As is usually done with generic relationships, the same name (namely, materialization) is used for both the generic relationship and its concrete realizations in applications.

Object-oriented programming languages have dominated the early years of object technology and their influence still clearly permeates the area. Their treatment of relationships as little more than pointer-valued attributes has confined relationships to a second-class status in most database management systems and, to a lesser extent, in software development methods. Thus, except for generalization, classification, and a simple version of aggregation, usual object models do not directly support generic relationships. Consequently, users are left with ad hoc implementation techniques, like pointers or references, with problems of dispersion and duplication of information among several participants.

An approach to implementing generic relationships with a metaclass mechanism was demonstrated in [14], where the part-of relationship was implemented in VODAK, an open object database management system [15]. In an object model, classes describe the structure and behavior of their instances with attributes and methods,

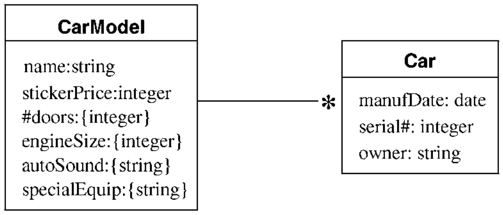

Fig. 1. An example of materialization.

respectively. Metaclasses [16] play the same role for classes: They describe the structure and behavior of classes with class variables and methods.

The metaclass construct allows for capturing the structure and behavior associated with a generic relationship R independently of specific application classes participating in the relationship. The semantics of R is defined once and for all in a metaclass R-Metaclass and an R-link between application classes is established by declaring them as instances of R-Metaclass. From a modeling point of view, the metaclass mechanism thus extends the object model with a new relationship R, which is made available to tailor classes to application needs.

This paper follows such an approach for implementing materialization. A metaclass AbstractConcreteClass is defined to capture the generic semantics of classes and objects participating in materializations. The metaclass defines methods for creating, deleting, and querying materialization links among application classes; it also helps create and delete instances of application classes conforming with the semantics of materialization. The metaclass also provides for attribute-propagation mechanisms associated with materialization.

The rest of the paper is organized as follows: Section 2 presents materialization. Section 3 surveys the metaclass mechanism, which plays a central role in our implementation. Section 4 characterizes the facilities required of the target system to support our implementation of materialization. Sections 5, 6, and 7 present, in detail, the metaclass semantics of materialization at both the class and the instance levels and several aspects of an abstract implementation, including attribute propagation. Section 8 summarizes and concludes the paper.

2 Materialization

This section gives an overview of materialization that elaborates on the presentation in [4].

2.1 Intuitive Definition

Intuitively, materialization relates a class of categories to a class of more concrete objects analyzed with those categories. Fig. 1 shows a materialization linking classes CarModel and Car. CarModel is the more abstract[1] class and Car is the more concrete class of materialization. A materialization link is drawn as a straight line with a $*$ on the side of the more concrete class.

1. The notion of abstractness/concreteness of materialization captures domain semantics. It is distinct from the system notion of abstract class in object models, where an abstract class is a class without instances whose complete definition is typically deferred to subclasses.

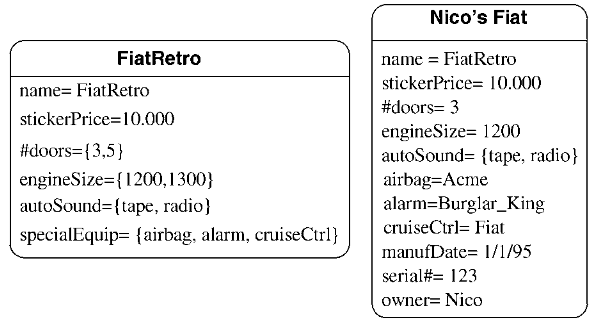

Fig. 2. Instances of CarModel and Car from Fig. 1.

CarModel represents information typically displayed in the catalog of car dealers, namely, name and price of a car model and lists of options for number of doors, engine size, sound equipment, and special equipment. Class Car represents information about individual cars, namely, manufacturing date, serial number, and owner identification.

Fig. 2 shows an instance FiatRetro of CarModel and an instance Nico's Fiat of Car of model FiatRetro. Intuitively, the materialization CarModel—$*$Car expresses that every concrete car (e.g., Nico's Fiat) has exactly one model (e.g., FiatRetro), while there can be any number of cars of a given model.

Further intuition about abstractness/concreteness is that each car is a concrete realization (or *materialization*) of a given car model of which it "inherits" a number of properties in several ways:

- Nico's Fiat directly inherits the name and sticker Price of its model FiatRetro; this mechanism is called Type 1 attribute propagation or T1 propagation for short.
- Nico's Fiat has attributes #doors, engineSize, and autoSound whose values are selections among the options offered by multivalued attributes with the same name in FiatRetro; this is called Type 2} (or T2) attribute propagation. For example, the value {1200,1300} of engineSize for FiatRetro indicates that each FiatRetro car comes with either engineSize = 1200 or engineSize = 1300 (e.g., 1200 for Nico's Fiat). Thus, the {1200,1300} value of engineSize for FiatRetro serves as domain, or type, for the engineSize attribute of a subclass of class Car consisting of cars with model FiatRetro.
- The value {airbag, alarm, cruiseCtrl} of attribute specialEquip for FiatRetro means that each car of model FiatRetro comes with three pieces of special equipment: an air bag, an alarm system, and a cruise-control system. Thus, Nico's Fiat has three new attributes named airbag, alarm, and cruiseCtrl, whose suppliers are, respectively, Acme, Burglar_King, and Fiat. Other FiatRetro cars may have different suppliers for their special equipment and cars of models other than FiatRetro may have a different set of pieces of special equipment. This mechanism is called Type 3 (or T3) attribute propagation.

In addition to attributes propagated from FiatRetro, Nico's Fiat of course has a value for attributes manufDate,

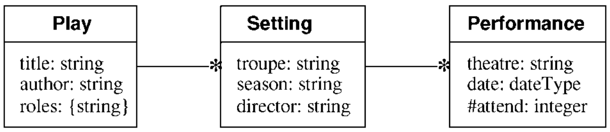

Fig. 3. Composition of materializations.

serial#, and owner of Car. The semantics of attribute propagation is defined in Section 2.2 and its implementation is described in Section 6.5.

Materializations can be involved in compositions where the concrete class of one materialization is also the abstract class of another one, etc. Fig. 3 shows a composition of two materializations. It deals with theater Plays with a title, an author, and a set of main roles. Plays materialize as Settings that add production decisions for a theatrical season: a troupe, a director, and a set of actors for each role. Settings materialize, in turn, as Performances, with a theatre where the performance takes place, a calendar date, the attendance #attend on that date, and with each role of Play assigned to a specific actor for each Performance. A class without more abstract class (like Play in Fig. 3) is called the root of the hierarchy, while a class without more concrete class (like Performance) is a leaf.

Fig. 4 shows an instance Ménage_à_Trois of Play, an instance Sett_Ma3_Fall98 of Setting, associated with Ménage_à_Trois, and an instance Perf_Ma3_051198 of Performance, associated with Sett_Ma3_Fall98. Ménage_à_Trois is an ordinary instance of Play (see Fig. 3). Sett_Ma3_Fall98 similarly holds values for the attributes of Setting; in addition, it inherits the value of attributes title and author of Ménage_à_Trois; it also creates three new attributes (husband, wife, and lover) from the value of roles in Ménage_à_Trois and it assigns them a domain ({Delon, Sharif}, {Bardot, Morgan}, and {Allen, Belmondo}, respectively) for their instances. Perf_Ma3_051198 holds values for the attributes of Performance (see Fig. 3), it inherits from Sett_Ma3_Fall98 the values of attributes title, author, troupe, season, and director, and it instantiates attributes husband, wife, and lover of Sett_Ma3_Fall98.

Abstract classes can materialize into several concrete classes. For example, data for a movie-rental store could involve a class Movie, with attributes director, producer, and year, that independently materializes into classes VideoTape and VideoDisc (i.e., VideoTape—$*$ Movie—$*$VideoDisc). VideoTapes and VideoDiscs could have attributes like inventory#, system (e.g., PAL, NTSC for VideoTape),

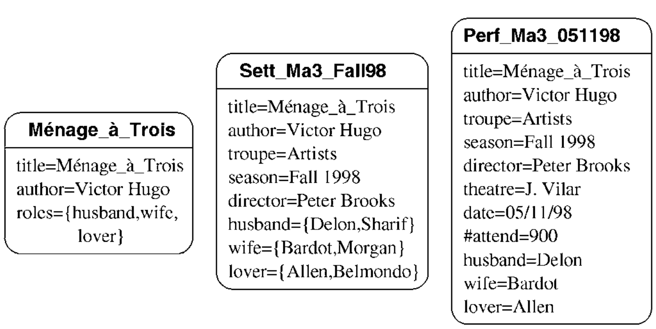

Fig. 4. Instances of Play, Setting, and Performance from Fig. 3.

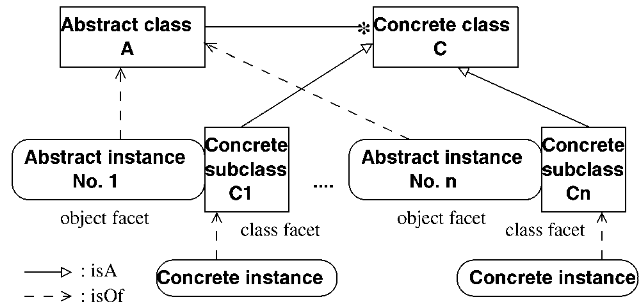

Fig. 5. Semantics of materialization.

language, availability (i.e., in-store or rented), and so on. This paper only considers tree-structured materialization hierarchies, i.e., we do not address concrete classes materializing more than one abstract class as in A—$*$C—$*$B. Those more general hierarchies involve a version of multiple inheritance.

2.2 Precise Semantics

We now summarize the necessary elements for a formal definition of materialization. Materialization is a binary relationship between two classes A and C, where A is more abstract than C (or C is more concrete than A). Abstractness/concreteness is a user-specified partial order consistent with the cardinalities and the attribute-propagation mechanisms of materialization.

Most real-world examples of materialization have cardinality $[1, 1]$ on the side of the concrete class C and cardinality $[0, n]$ on the side of the abstract class A. Application semantics can further constrain the cardinality on the A-side to $[c_{min}, c_{max}]$, meaning that at least c_{min} and at most c_{max} concrete objects are associated with each abstract object.

2.2.1 Two-Faceted Constructs

The semantics of materialization is conveniently defined as a combination of the usual *isA* (generalization) and *isOf* (classification) generic relationships and of a class/metaclass correspondence, as shown in Fig. 5. As in Figs. 1 and 2, we draw classes as rectangular boxes and instances as boxes with rounded corners. Classification links (*isOf*) appear as dashed arrows and generalization links (*isA*) as solid arrows.

In a system with metaclasses, a class can also be seen as an object. *Two-faceted constructs* make that double role explicit. Each two-faceted construct is a composite structure comprising an object, called the *object facet*, and an associated class, called the *class facet*. To underline their double role, we draw a two-faceted construct as an object box adjacent to a class box.

The semantics of materialization A—$*$C is expressed with a collection of two-faceted constructs as follows: Each object facet is an instance of abstract class A, while the associated class facet is a subclass of concrete class C. Materialization induces a partition of C into a family of subclasses $\{C_i\}$, such that each C_i is associated with exactly one instance of A. Subclasses C_i inherit attributes from C through the classical inheritance mechanism of the *isA* link. They also "inherit" attributes from A, through the mechanisms of attribute propagation described in the next section.

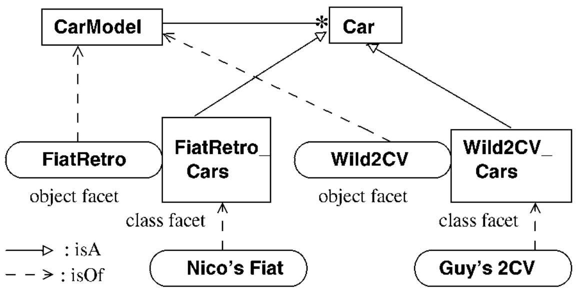

Fig. 6. Semantics of the materialization in Fig. 1.

Objects of C, with attribute values "inherited" from an instance of A, are ordinary instances of the class facet associated with that instance of A.

Of course, only application classes, like A and C (e.g., CarModel and Car), appear in conceptual schemas. The two-faceted construct machinery is managed by the implementation described later and is invisible to users. For them, attribute propagation is built-in and instances of application classes, like Nico'sFiat in Fig. 2, come with attribute values propagated from their abstract instances through materialization links.

Fig. 6 sketches the semantics of the materialization of Fig. 1. FiatRetro, an instance of CarModel, is the object facet of a two-faceted construct, whose class facet is FiatRetro_Cars, a subclass of Car, describing all instances of Car with model FiatRetro. Wild2CV is another instance of CarModel and Guy's 2CV is an instance of class facet Wild2CV_Cars. For users, Nico's Fiat and Guy's 2CV are instance of Car, with an instantiation mechanism that integrates attribute propagation, just like instantiation in object models with generalization integrates inheritance from a superclass to its subclasses. In our semantics and its implementation, Nico's Fiat and Guy's 2CV are instances of FiatRetro_Cars and Wild2CV_Cars, respectively.

Similarly, Fig. 7 illustrates the semantics of a composition of two materializations by displaying one two-faceted construct for each one. Ménage_à_Trois is an instance of Play. For users, Sett_Ma3_Fall98 and Perf_Ma3_051198 are instances of Setting and Performance, respectively. Our semantics describes them as instances of class facets Settings_of_Ma3 and Perfs_Ma3_Fall98, respectively.

2.2.2 Attribute Propagation

Objects of the concrete class naturally "inherit" information from objects of the abstract class, as illustrated in Section 2.1. We use, from now on, *attribute propagation* for the mechanisms associated with materialization and reserve *inheritance* for the usual propagation mechanism of attributes and methods from a superclass to its subclasses in a generalization.

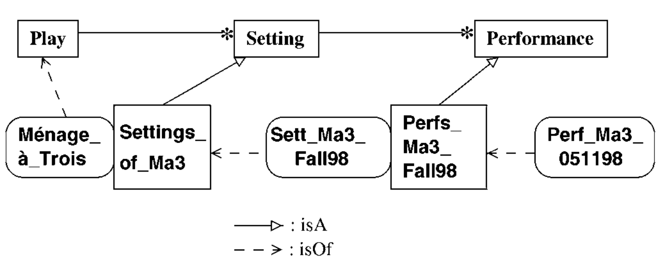

Fig. 7. Semantics of the materialization in Fig. 3.

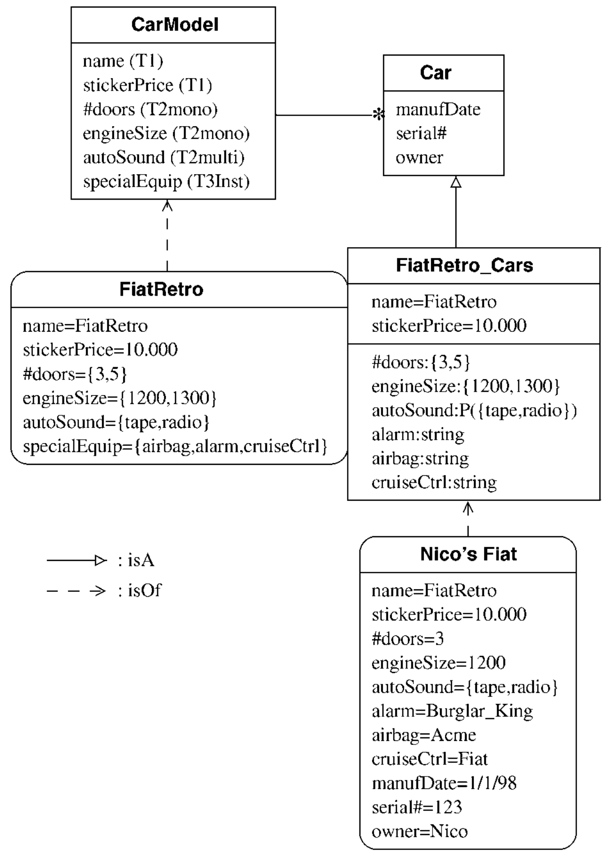

Fig. 8. Attribute propagation between CarModel and Car.

Attribute propagation with materialization is precisely defined as a transfer of information from an abstract object to its associated class facet in a two-faceted construct, as illustrated in Figs. 8 and 9. For clarity, attribute-propagation types are shown on abstract classes, although that information really belongs to the materialization links. The implementation described in the following sections indeed stores the propagation information separately from application classes (see, e.g., Figs. 15 and 16).

The following definitions on attributes will be useful. A *class attribute* of a class C has the same value for all instances of C; an *instance attribute* of C has its value defined for each instance of C. Attributes can be *monovalued* (i.e., their value is a single atomic value) or *multivalued* (i.e., their value is a set, possibly empty or singleton, of atomic values).

T1 propagation. For users, this mechanism characterizes the plain transfer of an attribute value from an instance of the abstract class to associated instances of the concrete class. In our semantics, the value of a (monovalued or multivalued) attribute is propagated from an object facet to its associated class facet as a class attribute (i.e., its value is the same for all instances of the class facet). For example, the value of the monovalued attributes name and stickerPrice (FiatRetro and 10.000, respectively) in object facet FiatRetro propagates as a value of class attributes with the same name in class facet FiatRetro_Cars (see Fig. 8). The mechanism is identical for multivalued attributes.

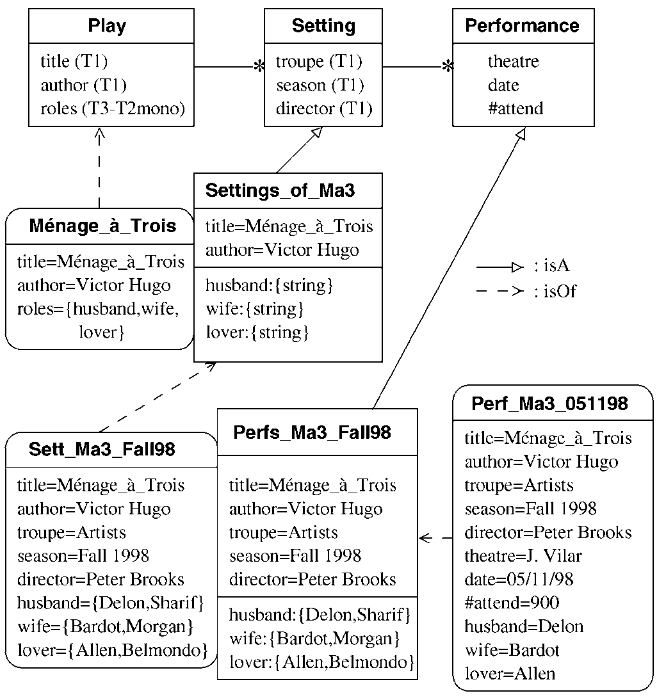

Fig. 9. Attribute propagation in Play—∗Setting—∗Performance.

T2 propagation. This mechanism concerns multivalued attributes of the abstract class A. For users, their value for an instance of A determines the domain (or type) of instance attributes with the same name, monovalued or multivalued, in the concrete class C. The associated propagation types will be named T2mono and T2multi, respectively. Again, our semantics goes through abstract objects and associated class facets.

An example of T2mono propagation is exhibited by engineSize, a multivalued attribute of CarModel (see Fig. 8). Its value, noted engineSize = {1200,1300}, for the FiatRetro object facet is the domain of values for a monovalued instance attribute with the same name in the associated class facet FiatRetro_Cars, where this is noted engineSize : {1200,1300}. Thus, each FiatRetro car comes either with engineSize = 1200 or with engineSize = 1300.

An example of T2multi propagation is exhibited by autoSound, a multivalued attribute of CarModel. Its value {tape, radio} in object facet FiatRetro indicates that each FiatRetro car comes with either tape or radio or both or nothing at all as autoSound. The associated class facet FiatRetro_Cars has a multivalued instance attribute autoSound with the powerset $\mathcal{P}(\{\text{tape}, \text{radio}\})$ as its domain.

T3 propagation. This mechanism is more elaborate. It also concerns multivalued attributes of the abstract class, whose value is always a set of strings. Each element in the set value of an attribute for an object facet generates a new instance attribute in the associated class facet. The domain of generated attributes must be specified in the definition of the materialization.[2]

For example, attribute specialEquip of CarModel propagates with T3 to Car (see Fig. 8). Its value {airbag, alarm, cruiseCtrl} for object facet FiatRetro generates three new monovalued instance attributes of domain string, named airbag, alarm, and cruiseCtrl, in the associated class facet FiatRetro_Cars. This propagation type will be called T3-Inst; it is the only possible T3 propagation for simple materializations.

Further propagation. For a composition A—∗C—∗D of two materializations, attributes propagated from A to C via A—∗C further propagate to D via C—∗D.

Attributes that propagate from A with T1 are class attributes in C and, thus, also in D.

Attributes that propagate from A with T2 or T3 produce instance attributes in C. If the latter are monovalued, they propagate with T1 to D. If they are multivalued, then they can propagate with T1, T2, or T3 to D.

For example, attribute roles of Play propagates with T3 in Play—∗Setting and generates multivalued attributes of domain {string} in Setting. The latter propagate with type T2mono in Setting—∗Performance, producing monovalued instance attributes in Performance (see Fig. 9). Thus, by T3 propagation, three new multivalued attributes are generated in class facet Settings_of_Ma3 from the value {husband, wife, lover} of attribute roles in the Ménage_à_Trois instance of Play. Their value in an instance Sett_Ma3_Fall98 of Settings_of_Ma3 is a set of names of actors available for playing each of the husband, wife, and lover roles during a specific theater season (namely, Fall 1998). Then, by T2 propagation, class facet Perfs_Ma3_Fall98 has three monovalued instance attributes, named husband, wife, and lover, whose domain is the value of the corresponding attribute in Sett_Ma3_Fall98. Finally, one among the actors available for each role is chosen for each performance (e.g., Delon as husband on 05/11/98 as shown in Perf_Ma3_051198). This propagation type of roles from Play to Setting and to Performance will be noted T3-T2mono, with the hyphen signaling two-level propagation. Other two-level propagation types work as expected.

The implementation presented in Section 6 accounts for the following propagation types: T1, T2mono, T2multi, and T3Inst for simple materializations and T3-T2mono and T3-T2multi for compositions of two materializations.

2.2.3 *Other Attribute-Propagation Types*

Case studies have suggested that other propagation types can be useful.

For example, in the CarModel—∗Car example of Fig. 8, attribute specialEquip of CarModel could propagate to Car with another propagation type. Its value {airbag, alarm, cruiseCtrl} for FiatRetro could be a list of optionally available pieces of equipment, in the T2 style, for FiatRetro cars. Each FiatRetro car would then come with a subset of {airbag, alarm, cruiseCtrl}, each with a manufacturer name, in the T3 style.

More general propagation mechanisms can be imagined to mimic the manipulation of information by stakeholders in application domains. Thus, a case study with complex data structures in molecular biology [17] has suggested elaborate propagation mechanisms that directly reflect reasoning patterns routinely practiced by molecular biologists. In these patterns, instances of the abstract class govern not only the propagation of values, as in the examples discussed so far,

2. For the sake of clarity, this domain is not shown in Figs. 8 and 9. See attribute genAttrType in Section 6.2 and Fig. 14.

but also the specific selection of substructures in the concrete objects. Equivalent models with simpler propagation mechanisms require extra classes specifically dedicated to providing explicit access paths for the transfer of information, but not otherwise needed in the application domain nor corresponding to naturally identified concepts.

Obviously, such richer propagation mechanisms, as well as T1, T2, and T3 propagation spanning a cascade of more than two materializations, become difficult to apprehend and use effectively. As often in conceptual modeling, the tradeoff is between directly capturing concepts and their associations, like attribute propagation, as they are perceived in the application domain, and managing the complexity of both their specification by modelers and their implementation with the currently available technology.

2.3 More Examples of Materialization

Materialization provides an extra degree of freedom for building conceptual schemas directly reflecting concepts that are natural in application domains. In summary, for class C to materialize class A (A—∗C), C must be more concrete than A in the partial order expressing abstractness. Materialization induces a partition of C into a family of subclasses, each associated with exactly one instance of A. Cardinalities must be $[1, 1]$ on the side of C and $[0, n]$ or tighter (i.e., $[c_{min}, c_{max}]$) on the side of A. Instances of materialization are ubiquitous, as illustrated by the following examples.

- Modeling air travel can involve a concept of itinerary (from an origin to a destination, with a distance, etc.), materialized as a class of flights (for an airline, with a price, on certain days of the week, periods of the year, etc.), itself materialized as a class of flights for specific calendar days (with a date, an aircraft, a crew, etc.).
- News items can materialize as articles in a particular edition of a newspaper, in turn materialized as physical copies of the newspaper.
- Stories can materialize as book titles (e.g., in publisher catalogs) that materialize as book copies (e.g., in library inventories). Stories can also materialize as theater plays, themselves materialized as performances, a variant of the example presented above (see Fig. 3). Movies are another materialization of stories; they can, in turn, materialize as video titles that can materialize as physical tapes and discs available in a video-rental store. Books in a library or in a bookstore can also be classified according to literary genre (e.g., drama, reference, travel). In a library, they can differ by their borrowing status (e.g., duration, price, reader privileges).
- For our running example with car models and cars, class CarModel can materialize as both brochures and videos presenting the models.
- Film negatives can materialize as positive prints, differing in size, shades of colors, etc.
- Sources for text formatters (e.g., LaTeX, HTML) materialize into printed versions of documents of various sizes and shapes.
- Forms (e.g., income-tax forms) materialize as filled-in forms (e.g., income-tax returns).
- Constitutions materialize into laws, in turn materialized as operational regulations.

Thus, materialization expresses various nuances of meta-information. The most common relationship is *classification* between categories and concrete objects classified with those categories. Materialization also frequently characterizes *embodiment*, the relationship between classes of objects and their common abstract definition, with the possibility of the relationship being associated to a *transformation* of objects to produce more detailed objects. In [4], we also introduce the materialization of relationships (e.g., aggregation) and the materialization of constraints.

2.4 The Power of Materialization

With T3 propagation, class facets and, consequently, concrete objects of a materialization in general do not have the same structure. For example, while class FiatRetro_Cars has attributes airbag, alarm, and cruiseCtrl, class Wild2-CV_Cars could have alarm and pwrSteer as corresponding attributes if the related Wild2CV model has {alarm, pwrSteer} as a value for attribute specialEquip. This heterogeneity of instances is not fundamentally different from that allowed in the instances of the subclasses of a common superclass with generalization.

Materialization also opens the door to more dynamic conceptual schemas. Suppose, for example, that a new engine size, say 1400, becomes available for cars of model FiatRetro. With T2 propagation, this simple update to the value of attribute engineSize in an instance of CarModel requires changing the definition of class facet FiatRetro_Cars so that the new domain of its engineSize attribute includes value 1400. With T3 propagation, schema updates triggered by updates to abstract instances are still more drastic. For example, adding a new piece of special equipment to model FiatRetro requires creating a new attribute for FiatRetro_Cars and for some instances of Car.

This extra complexity (heterogeneous instances for a class, dynamic schemas) required of system software is to be weighed against the convenience of flexible attribute-propagation mechanisms and against the increase in power and flexibility thus made available for modeling application domains.

2.5 Related Work on Materialization

The interest in capturing the semantics of materialization has been intuitively perceived, with different names, for as many as 20 years, according to [18]. Unlike our own work, little research has addressed the definition of a generic relationship for extending conceptual-modeling languages with that semantics. Also, the works referenced in this section view abstractness and concreteness as essentially absolute properties, unlike our own work, which treats abstractness/concreteness as a partial order.

Two constructs related to materialization are mentioned in OMT [19] and referred to as *metadata* and *homomorphisms*.

The term *materialization* was introduced and characterized informally in [20]. Our nearly formal presentation in [4] subsumed that definition. Since then, we have been refining the semantics of materialization and have implemented it in various settings [21], [22], [23].

The *power types* of [24] also catch the basic idea: "A power type is an object type whose instances are subtypes of another object type." A power type is, in our terms, the abstract class of a materialization, while the "other object type" is its concrete class. We find it more appropriate to attach the semantics to the relationship than to the abstract class. Also, our two-faceted constructs clearly show that, even if they are tightly related, the instances of the power type (the object facets) are not the same as the subtypes of the concrete class (the class facets). Indeed, our semantics implements attribute propagation, not discussed in [24], as taking place between an object facet and its associated class facet.

A semantics similar to that of power types is described as a "type object" design pattern in [25].

A knowledge level for object types and an operational level for objects are distinguished in [26], in the spirit of materialization.

Several examples of what we call materialization are presented in [27]. The two directed mappings equivalent to the relationship are referred to as "is an example of" and "is embodied in."

A similar semantics is analyzed as the "intension" and "extension" of concepts in [28], [29].

The use of classification is also advocated as a semantic constructor for conceptual models in [30], in the same spirit as materialization.

Several patterns frequently occurring in the real world are described in [31]; the closest to materialization is called *item-description pattern*.

A library of generic relationships for analysis is suggested in [11]; the *reference* association is closest to materialization, but it is defined somewhat informally and without attribute propagation.

Materialization has been shown to smoothly integrate an intensional view of taxonomies, based on a hierarchy of concepts structured by classification/instantiation, and an extensional view, based on a hierarchy of subclasses of the total population of objects structured by generalization [32], [33].

3 METACLASSES

This section presents the metaclass mechanism and its role in our implementation of materialization.

3.1 Various Metaclass Systems

In object models, metaclasses define the structure and behavior of classes, like classes define the structure and behavior of their instances. Systems with metaclasses comprise at least three levels: token (uninstantiable object), class, and metaclass. Additional levels, like Metaclass in Fig. 10, can be provided as root for the common structure and behavior of all metaclasses. The number of levels of such hierarchies varies from one system to another.

Substantial differences appear in the literature about the concept of metaclass. We suggest the following criteria to account for the variety of definitions [16]:

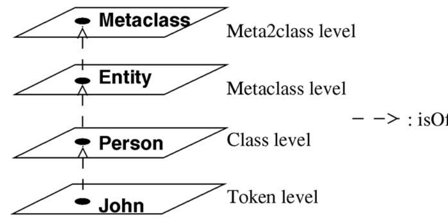

Fig. 10. Levels of systems with a metaclass concept.

1. **Explicitness.** The ability for programmers to explicitly declare a metaclass like they do for ordinary classes. Implicit metaclasses are automatically created by the system. Of course, explicit metaclasses are more flexible. They can, for example, be specialized into other metaclasses like ordinary classes. Explicit metaclasses are supported by several semantic models (e.g., TAXIS [34], SHM [35]), object models and systems (e.g., VODAK [15], ConceptBase [36]), knowledge-representation languages (e.g., KEE [37], Telos [38]), and programming languages (e.g., CLOS [39], Logtalk [40], Classtalk [41]). On the contrary, Smalltalk [42] and Gemstone [43], for example, support implicit system-managed metaclasses only.
2. **Uniformity.** The ability to treat an instance of a metaclass like an instance of an application class. Thus, for example, in Fig. 10, to create Entity, message new is sent to Metaclass; to create Person, the same message new is sent to Entity; and, again, to create object John, message new is sent to its Person class. While most metaclass systems support uniformity, Smalltalk, for example, does not.
3. **Depth of instantiation.** The number of levels for the hierarchy of classes and metaclasses. While, for example, Smalltalk has a limited depth in its hierarchy of metaclasses, VODAK and CLOS allow for an arbitrary depth.
4. **Circularity.** The ability to use metaclasses in a system for a uniform description of the system itself. To ensure finiteness, some metaclass concepts have to be instances of themselves. CLOS and ConceptBase, for example, offer that ability. Smalltalk does not.
5. **Shareability.** The ability for more than one class to share the same user-defined metaclass. Most systems supporting explicit metaclasses provide shareability.
6. **Applicability.** Whether metaclasses can describe classes only (the general case) or other concepts also. For example, TAXIS extends the use of metaclasses to procedures and exceptions, while ConceptBase uses attribute metaclasses to represent the common properties of a collection of attributes.
7. **Expressiveness.** The expressive power made available by metaclasses. In most systems, metaclasses can directly represent the structure and behavior of their instances only. On the other hand, a comprehensive semantics of materialization and of most generic relationships R concerns both classes and their instances (see Fig. 11). It is thus convenient that metaclasses implementing the semantics of R be able

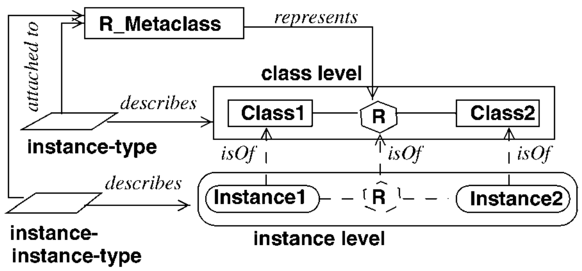

Fig. 11. Interaction between a relationship R and the metaclass defining it.

to deal with both the class level and the instance level in a coordinated manner. Thus, as done in VODAK [14], [44], it is natural to describe the semantics of R at both levels: An instance-type provides structure and behavior for the instances of the metaclass while an instance-instance-type provides structure and behavior for the instances of the instances of the metaclass. The metaformulas of Telos and ConceptBase can also specify the behavior of the instances of a metaclass and of the instances of its instances.

3.2 Usage of Metaclasses

Various reasons warrant a metaclass mechanism in a model or a system. Typically, metaclasses extend the system kernel, blurring the boundary between users and implementors. Explicit metaclasses can specify knowledge to:

- represent group information that concerns a set of objects as a whole. For example, the average age of employees is naturally attached to an EmployeeClass metalevel;
- represent class properties unrelated to the semantics of instances, like the fact that a class is concrete or abstract,[3] has a single instance or multiple instances, has a single superclass or multiple superclasses;
- customize the creation and the initialization of new instances of a class;
- enhance the extensibility and the flexibility of models and, thus, allow easy customization. For example, new behavior can be introduced by controling method execution through reflection [45], [46], [47] (see also [48] for a report on application experience with ConceptBase);
- directly capture the semantics of generic relationships, like we do for materialization in this paper.

3.3 Metaclass Approaches to Deal with Materialization

Fig. 12 illustrates three metaclass approaches for implementing materialization. They are discussed in more detail in [3]. Another strategy for implementing generic relationships with a reflective approach in CLOS is reported in [46].

The two-metaclass approach (see Fig. 12a) consists of defining two metaclasses, AbstractClass and Concrete Class, for the roles of abstract class and concrete class, respectively. Two types are associated to each metaclass, accounting for the class and instance-level semantics, respectively.

3. Here, an abstract class, in the usual sense of object models, is an incompletely defined class without direct instances whose complete definition is deferred to subclasses.

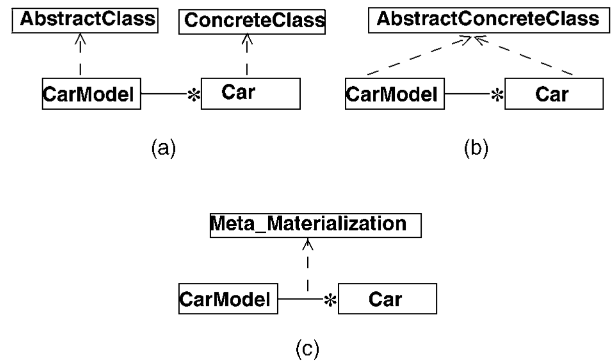

Fig. 12. Metaclass approaches to materialization implementation. (a) The two-metaclass approach. (b) The single-metaclass approach. (c) The relationship-metaclass approach.

Thus, for CarModel—∗Car, CarModel and Car are created as instances of AbstractClass and ConcreteClass, respectively. They inherit from their metaclass attributes and methods enforcing the class and instance-level semantics of materialization.

The two-metaclass approach has been used to formalize materialization in the Telos modeling language [21].

The single-metaclass approach (see Fig. 12b) consists of defining a metaclass AbstractConcreteClass for both roles of materialization. As above, two types are associated with the metaclass accounting, respectively, for the class- and instance-level semantics.

This approach has been used to implement aggregation in VODAK [14], [49], and materialization in Logtalk [23]. The approach is advocated in [44] to implement all generic relationships. It is adopted for the implementation of materialization described in the following sections.

The relationship-metaclass approach (see Fig. 12c) consists of defining a metaclass Meta_Materialization for directly representing materialization links. Again, two types are associated with the metaclass to represent the class and instance-level semantics.

Thus, for example, CarModel and Car would be created as instances of a usual metaclass, while materialization CarModel—∗Car would be an instance of Meta_Materialization.

4 Facilities Required of the Target Object System

This section characterizes the facilities required of the target system for which our implementation of materialization is formulated. They are of two kinds: 1) metaclass support and 2) schema evolution, that is, the ability to dynamically change a database schema.

1. For metaclass support, our implementation assumes:
   - classical object-modeling facilities and abstraction mechanisms like classification and generalization;
   - a metaclass mechanism with the following properties, relating, respectively, to the criteria

of explicitness, shareability, and expressiveness, discussed in Section 3:

- the possibility of explicitly creating a metaclass (the semantics of materialization will be defined in an AbstractConcreteClass metaclass);
- the ability for several classes to share a user-defined metaclass (e.g., for a materialization CarModel—∗Car, both CarModel and Car will be instances of AbstractConcreteClass);
- the possibility of defining, in the same metaclass, an abstract data type for the instances of the metaclass and another type for the instances of its instances;

- a generic type for objects and attributes to serve as a place holder for classes and attributes, respectively, to be instantiated in applications.[4] It will act like a formal parameter in the parameterized metaclass describing the generic semantics and be substituted by actual classes and domains when application classes are created as instances of the metaclass.

2. For schema evolution, the target system is assumed to provide the possibility of dynamically creating and deleting a class as a subclass of a given class and of dynamically changing the domain of an attribute of a given class.

5 General Structure of Materialization Implementation

This section surveys our implementation of materialization, the central contribution of the paper. A metaclass AbstractConcreteClass is defined to capture the generic semantics of classes and objects participating in materializations through the definition of two abstract data types. At the class level, ACClass-InstType endows application classes with the means of defining and querying the materialization links existing between them; it also allows them to create and delete instances conforming to the semantics of materialization. At the instance level, ACClass-InstInstType provides the instances of instances of AbstractConcreteClass (i.e., the instances of application classes) with structure and methods for establishing, deleting, and querying materialization links. The metaclass also provides for the attribute-propagation mechanisms associated with materialization.

Section 6 presents the structure of ACClass-InstType (see Fig. 13), while Section 7 is devoted to ACClass-InstInstType (see Fig. 22). Detailed algorithms can be found in [22].

The definition and use of a generic relationship like materialization involves several stages which take place in order. The first stage is, of course, the definition of metaclass AbstractConcreteClass that embodies the generic semantics of the relationship. The metaclass is made available to applications as an extension of the data-definition mechanisms of the target system.

4. In untyped systems like CLOS, all variables are of a generic type. In a typed system, a generic type for objects and attributes would have to be explicitly available.

```
Define type ACClass-InstType
  Attributes
    theMatRelshps: {matRelationshipType}
    theAbstractClass:OID

  Methods
    defConcreteRelshps(someRelshps: {matRelationshipType});
    defAbstractClass(aClass: OID);

    makeAbstractObject(): OID;
    makeConcreteObject(anObjectFacet: OID): OID;
    destroy(anObject: OID): BOOL;

    getMinCardinality (anAC_Class: OID): integer;
    getMaxCardinality (anAC_Class: OID): integer;
    isAbstractClassOf(anAC_Class: OID): BOOL;
    isConcreteClassOf(anAC_Class: OID): BOOL;
    getConcreteClasses(): {OID};
    getAbstractClass(): OID;
    getInhAttribT1(anAC_Class: OID): {Attribute1-Def};
    getInhAttribT2(anAC_Class: OID): {Attribute2-Def};
    getInhAttribT3(anAC_Class: OID): {Attribute3-Def};
END
```

Fig. 13. Interface of ACClass-InstType.

An application can then invoke the generic template by making application classes, like A and C, that are to participate in materialization A—∗C, instances of the metaclass (see Fig. 15a). Classes like A and C are both referred to as AbstractConcrete (or AC) classes.

Upon instantiation of the metaclass, information describing the specific characteristics of materialization A—∗C must also be provided by the schema designer. That information includes the cardinality at the A side and the characteristics of attribute propagation from A to C. Methods defConcreteRelshps and defAbstractClass (see Fig. 15b) establish the link between A and C and initialize structures used for creating instances and for querying AC classes about their materialization characteristics.

Methods makeAbstractObject and makeConcreteObject of ACClass-InstType are then available for creating instances of application classes. As explained in Sections 6.3 and 6.4, when an abstract instance is created by makeAbstractObject, its corresponding class facet is also created and attribute propagation from the object facet to the class facet takes place. Because a concrete object cannot exist without a related abstract object that it materializes, method makeConcreteObject, at the same time that it creates a new concrete object, also creates a materialization link for the new object.

Method destroy of ACClass-InstType deletes abstract and concrete objects after suppressing the materialization links in which they participate.

Finally, ACClass-InstInstType also supplies methods to establish, delete, and query materialization links between abstract and concrete objects.

An important design decision was to attach the characteristics of materialization to its abstract class. This is certainly more natural than attaching the information to the concrete class since instances of the abstract class can exist without related concrete instances. Another solution could have been to implement materialization as separate structures, distinct from the participating classes.

Appealing as it looks, the distinction between methods for application classes in ACClass-InstType and methods for the instances of application classes in ACClass-InstInstType is

```
DATATYPE matRelationshipType= [
    theConcreteClass: OID,
    cardinality: [min:integer, max:integer],
    inhAttribT1: {Attribute1Def},
    inhAttribT2: {Attribute2Def},
    inhAttribT3: {Attribute3Def}]

DATATYPE Attribute1Def= [attrName: string]
DATATYPE Attribute2Def= [attrName: string,
          derivedAttr: {mono,multi}]
DATATYPE Attribute3Def= [attrName: string,
  genAttrType: TypeDef, genAttrPropag: {Inst,T2mono,T2multi}]
```

Fig. 14. matRelationshipType for ACClass-InstType.

not as clear-cut as for a relationship like part-of [14], where objects of a component class can exist without necessarily participating in an instance of part-of. The tighter semantic connection between abstract and concrete objects in materialization requires that creation and deletion of materialization links between application objects always happen in the context of creation and destruction of concrete objects. Thus, methods addConcreteObject, setAbstractObject, removeConcreteObject, and removeAbstractObject of ACClass-Inst-InstType (see Fig. 22) are never called independently of methods makeAbstractObject, makeConcreteObject, and destroy of ACClass-InstType. They are private methods hidden from users.

6 Class-Level Semantics: AbstractConcreteClass Instance Type

6.1 General Structure

Type ACClass-InstType endows the instances of Abstract ConcreteClass (i.e., application classes like CarModel and Car) with structure and behavior consistent with the semantics of materialization. Fig. 13 shows the interface of ACClass-InstType as composed of two parts: Attributes and Methods.

ACClass-InstType defines two attributes: theMatRelshps and theAbstractClass. The former is a set of matRelationshipType structures, each describing characteristics of a specific materialization (the concrete class and the propagation types for attributes of the abstract class, as shown in Fig. 14). For instance, if class A materializes in two classes B and C (i.e., B—*A—*C), then theMatRelshps associated with A will contain two structures describing the characteristics of materializations A—*B and A—*C, respectively. The second attribute, theAbstractClass, identifies the abstract class corresponding to a given concrete class.

The methods of ACClass-InstType provide the following functions:

- definition of the specific characteristics of a materialization and declaration of the abstract and concrete roles for AC classes with methods defConcreteRelshps and defAbstractClass;
- creation of abstract objects; when the constructor method makeAbstractObject creates an object, it also creates its associated class facet and it propagates attributes of the abstract object into the class facet;

```
CLASS CarModel InstanceOf              CLASS Car InstanceOf
        AbstractConcreteClass                  AbstractConcreteClass
    Attributes                             Attributes
        name: string;                          manufDate: date;
        stickerPrice: integer;                 serial# :integer;
        #doors:{integer};                      owner: string;
        engineSize:{integer};              Methods
        autoSound: {string};                   setSerial#(integer)
        specialEquip: {string};                setManufDate(date)
    Methods                                    ...
        setName(string);               END
        getName(): string
        get#doors():{integer}
        add#doors(integer)
        remove#doors(integer)
        ...
END
```

(a)

```
CarModel→defConcreteRelshps ({
    [theConcreteClass=Car,
    cardinality= [0,n],
    inhAttribT1= {name, stickerPrice},
    inhAttribT2= {[#doors, mono], [engineSize, mono],
        [autoSound, multi] },
    inhAttribT3= {[specialEquip, string, Inst]} ]
})
Car→defAbstractClass (CarModel)
```

(b)

Fig. 15. Definition of materialization CarModel—*Car. (a) Class declaration and (b) setting materialization characteristics.

- creation of concrete objects, with the constructor method makeConcreteObject and linking to their associated abstract object;
- deletion of objects of AC classes, with the destructor method destroy, consistently with the semantics of materialization;
- querying of AC classes about various aspects of their materialization relationship.

ACClass-InstType defines methods for both the abstract and the concrete classes. By making an AC class an instance of the metaclass, all methods are made available to the class, be it abstract or concrete. This decision simplified our implementation. Appropriate error messages are issued if incorrect method invocations are attempted.

The following sections examine, in more detail, the function of ACClass-InstType methods.

6.2 Semantics of Materialization Creation

Methods defConcreteRelshps and defAbstractClass define structures for storing the specific characteristics of a concrete materialization between application classes. Method defConcreteRelshps is applied to root abstract classes (i.e., classes without abstract class). Method defAbstractClass is applied to concrete classes (which can also be abstract in a composition of materializations) and specifies, in its aClass parameter, the abstract class of the target class.

The parameter someRelshps of defConcreteRelshps is a set of matRelationshipType structures (see Fig. 14) that specify the characteristics of all materializations in which the target class participates. The first field of matRelationshipType (theConcreteClass) is a place holder for the concrete class (the generic type for objects, noted OID, as in VODAK). It acts like a formal parameter to be substituted by an application class when the metaclass is instantiated (e.g., Car, in Fig. 15b). The abstract class need not explicitly appear since method defConcreteRelshps is necessarily applied to an abstract class. The second field of matRelationshipType (cardinality) specifies the cardinality at the abstract class side. The cardinality at the concrete class is not declared explicitly as it is always $[1, 1]$.

The remaining fields (inhAttribT1, inhAttribT2, and inhAttribT3) specify propagation types for attributes of the abstract class as follows:

- Attribute1Def is the name of an attribute propagating with T1;
- Attribute2Def gives, for an attribute propagating with T2, its name and the kind derivedAttr (monovalued or multivalued, corresponding to propagation types T2mono and T2multi, respectively) of the derived instance attribute;
- Attribute3Def gives the name of an attribute propagating with T3, a domain TypeDef, specified in genAttrType, for the generated attributes, and a propagation type genAttrPropag for the generated attributes. TypeDef is a place holder for the domain of attributes generated by T3 propagation. It acts like a formal parameter to be substituted by an actual domain when the metaclass is instantiated (e.g., string, in Fig. 15b).

Of course, the implementation checks that all attributes appearing in Attribute1Def, Attribute2Def, and Attribute3Def have been declared as attributes of the abstract class of the materialization.

Attributes generated with T3 can be ordinary instance attributes (T3Inst propagation), the only possibility for a simple materialization A—∗C. For a composition of materializations A—∗C—∗D, instance attributes in C propagate with T1 in C—∗D. A multivalued attribute of A can also propagate with T3 in A—∗C and generate in C multivalued attributes that propagate with T2 in C—∗D. The resulting attributes in D are instance attributes that can be monovalued (T3-T2mono propagation) or multivalued (T3-T2multi propagation). Section 6.5 describes the implementation of attribute propagation with the two-faceted machinery.

As an example, Fig. 15 shows how the CarModel—∗Car materialization is established by invoking the generic semantics. Both CarModel and Car are declared as instances of AbstractConcreteClass. The argument of defConcreteRelshps[5] specifies that: The concrete class related to CarModel is Car; the cardinality for CarModel is $[0, n]$; name and stickerPrice propagate with T1; #doors and engineSize both propagate with type T2mono, while autoSound propagates with type T2multi; specialEquip propagates with T3Inst, generating new instance attributes of domain string.

Note that method defAbstractClass is not called for a class that is the root of a materialization hierarchy (i.e., a class without abstract class), like CarModel in the example. Similarly, method defConcreteRelshps is not called for leaf classes (i.e., classes without more concrete classes), like Car in the example.

```
Play →defConcreteRelshps({
    [theConcreteClass=Setting,
    cardinality=[0, n],
    inhAttribT1={title, author}
    inhAttribT2={},
    inhAttribT3={[roles, {string}, T2mono]}]
})
Setting→defAbstractClass(Play)
Setting→defConcreteRelshps ( {
    [theConcreteClass=Performance,
    cardinality=[1, 4],
    inhAttribT1={troupe, season, director},
    inhAttribT2={},
    inhAttribT3={} ]
})
Performance→defAbstractClass (Setting)
```

Fig. 16. Definition of the composition of materializations of Fig. 9.

To define a composition of materializations, say A—∗C—∗D, both methods defConcreteRelshps and defAbstractClass must be invoked on class C as it is involved both as an abstract class and as a concrete class. Class A, the root of the hierarchy, only requires defConcreteRelshps, while class D, the leaf, only requires defAbstractClass.

Fig. 16 shows the corresponding method invocations for the composition of materializations Play—∗Setting—∗ Performance of Fig. 9. Thus, for example, in materialization Play—∗Setting, roles propagates with type T3-T2mono, as explained in Section 2.2.2. The implementation of attribute propagation is systematically defined in Section 6.5.

6.3 Creation of an Abstract Instance

Method makeAbstractObject creates a new instance of an abstract class which is the root of a hierarchy of materializations. In addition, makeAbstractObject} associates with the new object a class facet in which it propagates the attributes of the abstract class.

Specifically, for materialization A—∗C (see Fig. 17), makeAbstractObject, supplied with values for the attributes of A, creates a new abstract object a by invoking method new[6] on A. Object a becomes the object facet of a two-faceted construct, for which makeAbstractObject also creates a class facet Cf_a as a subclass of the concrete class C, for each materialization in which A participates. Cf_a comprises attributes inherited from the concrete class C by the *is-A* link and attributes InhAttr(A) propagated from the object facet a. Upon creation, class facet Cf_a has no instances: They will be created by subsequent calls of makeConcreteObject.

For example, the abstract object FiatRetro in Fig. 8 is created by makeAbstractObject. The structure of class facet FiatRetro_Cars, whose instances are cars of model FiatRetro, is created as a result of the same call of makeAbstractObject. One set of attributes of FiatRetro_Cars (i.e., manufDate, serial#, and owner) are inherited by subclassing. The other attributes of FiatRetro_Cars are propagated from the object facet FiatRetro: name and stickerPrice with type T1, #doors, and engineSize with

5. Method invocation is noted with the C^{++} syntax: an alias to the target object class, followed by "→ ", and by the method name and arguments.

6. Method new is supposed to be provided by the target object system to create instances of classes.

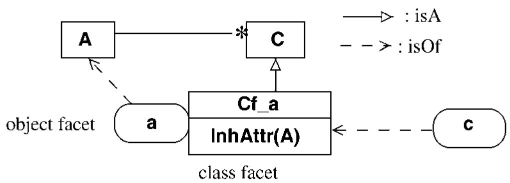

Fig. 17. Creation of abstract and concrete instances.

type T2mono, autoSound with type T2multi, while alarm, airbag, and cruiseCtrl propagate from specialEquip with type T3Inst. For optimization purposes, as will be seen in Section 6.5, only T2 and T3 propagation from CarModel leads to actual attribute creation in FiatRetro_Cars, while attributes propagated with T1 remain in FiatRetro where their value is accessed from instances of FiatRetro_Cars by the mechanism of delegation.

For a composition of materializations, like A—∗C—∗D in Fig. 18, the creation of instances of the root A of the hierarchy is realized as a simple materialization. However, as explained in the next section, the creation of an instance c of C is not done by method makeAbstractObject since c is first viewed as a concrete object in materialization A—∗C, before playing the role of abstract object in materialization C—∗D.

6.4 Creation of a Concrete Instance

Method makeConcreteObject (see Fig. 13) creates a new concrete object and relates it to an existing abstract object passed as parameter anObjectFacet. Note the asymmetry between makeAbstractObject and makeConcreteObject. While the former simply creates an instance of a root abstract class, the latter creates a concrete instance and links it to an existing instance of an abstract class. In effect, no concrete object can exist without an abstract object that it materializes.

Method makeConcreteObject invokes method new to create an object c as instance of class facet Cf_a associated with an object a. Although, for external users, c is an instance of the concrete class C, the implementation builds its structure from that of Cf_a since part of the structure of c originates from abstract object a and is implemented into Cf_a by attribute propagation.

For a simple materialization A—∗C, C is the leaf concrete class and the responsibility of makeConcreteObject is limited to creating the concrete object c.

For a composition of materializations, the concrete object may also be an abstract object for another more concrete class. In that case, its associated class facet must be created as a result of the same call of makeConcreteObject (see Fig. 18). As C is also abstract with respect to D, class facet Cf_c corresponding to c is created and attribute propagation is carried out from c to Cf_c. Thus, for that part of its work, makeConcreteObject performs a function similar to that of makeAbstractObject. Finally, to create the leaf concrete object d, makeConcreteObject behaves exactly as for a simple materialization.

For example, in the composition of materializations of Fig. 9, Ménage_à_Trois, an instance of Play, is created by makeAbstractObject. Sett_Ma3_Fall98, an instance of

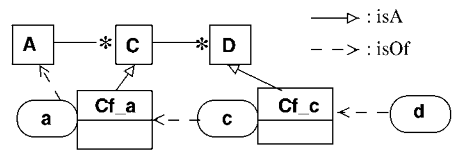

Fig. 18. Instance creation in a composition of materializations.

Setting, is created by the call Sett_Ma3_Fall98:=Setting → makeConcreteObject(Ménage_à_Trois). This call 1) makes Sett_Ma3_Fall98 a concrete object of Ménage_à_trois, 2) creates class facet Perfs_Ma3_Fall98 as a subclass of Performance, and 3) carries out attribute propagation from Sett_Ma3_Fall98 to Perfs_Ma3_Fall98.

6.5 Attribute Propagation

6.5.1 Attribute Propagation in a Simple Materialization

Consider Fig. 17 for the propagation of attributes from an abstract object a to its associated class facet Cf_a and its instances c.

T1 propagation does not physically propagate attributes from a to Cf_a and c. For example, the value of attribute name of Nico's Fiat is not stored in Nico's Fiat nor in FiatRetro_Cars, as suggested in Fig. 8. Instead, it is accessed in FiatRetro by delegation.

Usual message handling processes request o → attr by simply returning the value of attribute attr of object o (see Fig. 19). We modified message handling to handle T1 propagation. Request o → attr to concrete object o is delegated to o's abstract object, where the value of attr is stored. For a composition of materializations, the request is delegated iteratively until attr is found or the root object of the hierarchy is reached and it has no attr attribute. In Fig. 19, the usual and the extended message handler are invoked with "→ " and "→* ", respectively.

Using delegation for accessing attribute values propagated with T1 is solely motivated by optimization purposes, to avoid duplicating information in an object facet and in its corresponding class facet. Attribute values could instead be stored redundantly as class attributes in class facets and accessed with the standard message handler.

When creating the abstract instance a, method makeAbstractObject also creates the associated class facet Cf_a as a subclass of C. The additional attributes defined in Cf_a implement T2 and T3 propagation from a. They are made

```
Usual message handler
o→attr returns TypeDef {
  if object o is undefined
  then return "object o is undefined";
  if attr is in o's properties
  then return value of attr
  else return "attr is undefined" }

Extended message handler
o→*attr returns TypeDef {
  if object o is undefined
  then return "object o is undefined";
  if attr is in o's properties
  then return value of attr
  else return ( (o→getAbstractObject())→*attr ) }
```

Fig. 19. Extension of message handling for T1 attributes.

available to method makeAbstractObject through calls A $\rightarrow$ getInhAttribT2(C) and A $\rightarrow$ getInhAttribT3(C).

Thus, for example, the implementation of T2 propagation in materialization CarModel—$*$Car (see Fig. 15b) for attributes #doors, engineSize, and autoSound gives rise, in class facet FiatRetro_Cars, when it is created, to the following instance attributes, shown here with their domain:

$$\text{engineSize} : \{1200, 1300\};$$
$$\#\text{doors} : \{3, 5\};$$
$$\text{autoSound} : \mathcal{P}(\{\text{tape}, \text{radio}\});$$

By T3Inst propagation in CarModel—$*$Car, attribute specialEquip generates instance attributes of domain string. Its value {airbag, alarm, cruiseCtrl} in FiatRetro generates the following attributes in class facet FiatRetro_Cars:[7]

$$\text{airbag} : \text{string};$$
$$\text{alarm} : \text{string};$$
$$\text{cruiseCtrl} : \text{string};$$

6.5.2 Attribute Propagation in a Composition of Materializations

As for a simple materialization, T1 propagation does not explicitly generate attributes in class facets. Instead, attribute values are accessed by delegation. Attribute propagation from a root abstract object a to its associated class facet Cf_a works as simple materialization.

For a composition of materializations A—$*$ C—$*$ D, let attributes A1, A21, A22, and A3 be defined in class A with propagation types T1, T2mono, T2multi, T3, respectively. Let their value in instance a of A be as follows: A1 = u1, A21 = {v1,v2}, A22 = {w1,w2}, A3 = {x1,x2}. Attribute propagation works as follows (see Fig. 18):

- The value u1 of A1 is stored in a. It is a class attribute in Cf_a and Cf_c. Its value in a is accessed from the (in)direct concrete objects of a (e.g., c and d) by delegation.
- For attribute A21 of A, with value {v1,v2} in a, the derived monovalued attribute A21 has value v1 or v2 in c. Attribute A22 of A gives rise in Cf_a to a multivalued instance attribute with the same name and with some subset of {w1,w2} as value in c. A21 and A22 are class attributes in Cf_c. The value of A21 and A22 need not be stored in Cf_c as it will be accessed in c from d by delegation.
- For attribute A3 of A, with value {x1,x2} in a, x1 and x2 become the names of two new attributes in Cf_a. These new attributes have a common domain supplied in the definition of materialization A—$*$C (namely, genAttrType in structure Attribute-3Def of Fig. 14). The propagation of x1 and x2 to Cf_c works as follows:
  - For propagation type T3Inst, x1, and x2 become the name of instance attributes in Cf_a and of class attributes in Cf_c; their value in c need not be stored in Cf_c as it will be accessed in c from d by delegation;
  - For propagation type T3-T2mono, x1, and x2 become the names of multivalued attributes in Cf_a; their value in c becomes the domain of monovalued instance attributes with the same name (x1 and x2) in Cf_c;
  - Propagation type T3-T2multi works like T3-T2mono except that the attributes in Cf_c are multivalued.

To summarize, instances d of D have attribute values corresponding to all attributes propagated from A. Only those attributes of A propagating with types T3-T2mono and T3-T2multi have values physically stored in d.

For example (see Fig. 9), the value of attributes title and author of Play, propagating with T1, is accessed by Setting and Performance objects (i.e., instances of class facets Settings_of_Ma3 and Perfs_Ma3_Fall98) by delegation. For instance, when Perf_Ma3_051198 is asked its author, it delegates the request to its direct abstract object (i.e., Sett_Ma3_Fall98), which, in turn, queries its direct abstract object (i.e., Ménage_à_Trois), which returns the requested value (i.e., Victor Hugo).

Attribute roles in Play propagates with T3-T2mono. Its value {husband, wife, lover} in Ménage_à_Trois generates three multivalued attributes in class facet Settings_of_Ma3 that further propagate with T2mono; thus, from their value in Sett_Ma3_Fall98, the following three monovalued attributes are generated for class facet Perfs_Ma3_Fall98:

$$\text{Husband} : \{\text{Delon}, \text{Sharif}\};$$
$$\text{Wife} : \{\text{Bardot}, \text{Morgan}\};$$
$$\text{Lover} : \{\text{Allen}, \text{Belmondo}\};$$

6.6 Instance Deletion

Method destroy allows an AC class to delete its instances. Unlike the creation of instances, deletion can be accomplished with a single method. To delete an instance o of an AC class, destroy operates as follows:

- If o is a leaf concrete object (Fig. 20a), then delete o unless deletion violates the minimal cardinality with respect to its abstract object a;
- If o is an abstract object (Fig. 20b), then delete the instances of o's class facets (if any), delete the class facets, and finally delete object o, unless deletion violates some minimal cardinality.

For a composition of materializations, destroy operates as follows on an object o:

- If o is a leaf concrete object (Fig. 21a), then delete o unless deletion violates the minimal cardinality with respect to its abstract object a;
- Otherwise, o is an abstract object also concrete in a composition of materializations (Fig. 21b) or o is a root abstract object (Fig. 21c). Then, iteratively delete the instances (if any) of o's class facet, the class facet

7. More powerful modeling languages, like Telos or ConceptBase, can manage attributes directly at the metalevel. Then, for example, specialEquip in CarModel can be defined as a metatype with instance attributes airbag, alarm, and cruiseCtrl [21].

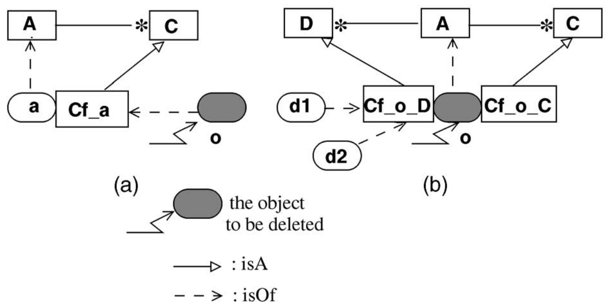

Fig. 20. Deleting objects in a simple materialization.

itself and, finally, delete object o, unless deletion violates some minimal cardinality.

6.7 Querying a Materialization Hierarchy

Materialization relationships can be queried with the access methods shown in Fig. 13.

Methods getMinCardinality and getMaxCardinality give, respectively, the minimal and the maximal cardinalities in the target class with respect to its concrete class passed in the anAC_Class parameter.

Methods isAbstractClassOf and isConcreteClassOf test whether the target class is a direct abstract (respectively, concrete) class corresponding to the concrete (respectively, abstract) class denoted by the anAC_Class parameter. Similarly, getAbstractClass and getConcreteClasses return the set of direct abstract (respectively, concrete) classes related to the target class. These methods query one level of materialization at a time.

The last three methods (getInhAttribT1/T2/T3) are provided to access propagation types of attributes. When an abstract class A materializes in several concrete classes C_i, attribute propagation is specified for each materialization A—$*C_i$. Thus, parameter anAC_Class specifies the concrete class to which attributes are propagated.

For example, the following queries can be addressed to materialization CarModel—$*$Car:

- CarModel→ getConcreteClasses() returns the concrete classes related to CarModel (i.e., {Car});
- CarModel→ getInhAttribT2(Car) returns the attributes propagated with T2 from CarModel to Car (i.e., [#doors,mono], [engineSize,mono], [auto Sound,multi]});
- CarModel→ getMinCardinality(Car) returns 0;
- CarModel→ isAbstractClassOf(Car) returns True;
- Car→ isConcreteClassOf(CarModel) returns True.

The following queries can be addressed to the composition Play—$*$Setting—$*$Performance:

- Play→ getConcreteClasses() returns the direct concrete class Setting;
- Setting→ getMinCardinality(Performance) returns 2 (assuming at least 2 performances for each play setting);
- Setting→ isAbstractClassOf(Performance) returns True;

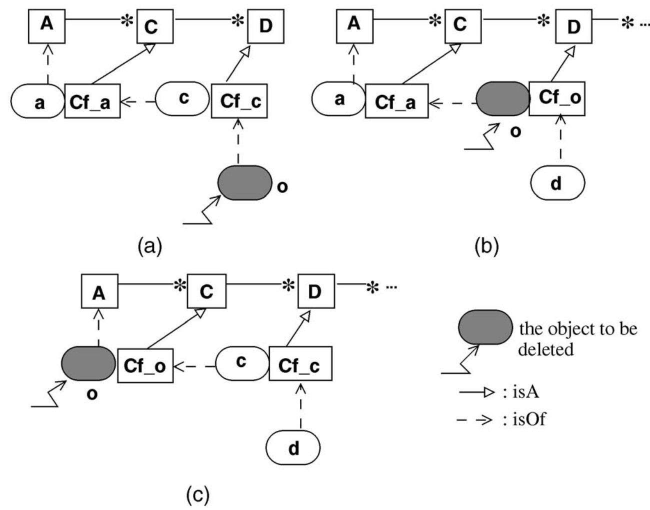

Fig. 21. Deleting objects in a composition of materializations.

- Play→ isAbstractClassOf(Performance) returns False as the method only takes into account the direct abstract class;
- Performance→ isConcreteClassOf(Play) returns False as the method only considers the direct concrete class.

7 Instance-Level Semantics: AbstractConcreteClass Instance Instance Type

This section describes the methods of ACClass-InstInst Type, whose interface is shown in Fig. 22. As discussed in Section 5, methods for creating and suppressing materialization links (i.e., addConcreteObject, setAbstractObject, removeConcreteObject, and removeAbstractObject) are invoked only in the context of object creation and destruction by methods of ACClass-InstType. They are private methods hidden from users.

ACClass-InstInstType has two attributes, theConcreteObjects and theAbstractObject. Since a concrete class has cardinality $[1,1]$ with respect to its abstract class, a concrete object is always associated with a single abstract object made explicit in attribute theAbstractObject. Attribute theConcreteObjects contains all concrete objects of an abstract object. They can be of different classes when the abstract class materializes in more than one concrete class.

```
Define type ACClass-InstInstType
  Attributes
      theConcreteObjects:{OID};
      theAbstractObject:OID;
  Methods
      Private
        addConcreteObject (aConcreteObject: OID) : BOOL;
        setAbstractObject (anAbstractObject: OID);
        removeConcreteObject (aConcreteObject: OID) : BOOL;
        removeAbstractObject (anAbstractObject: OID);
      Public
        getConcreteObjects () : {OID};
        getAbstractObject() : OID;
        class*() : OID;
END
```

Fig. 22. Interface for ACClass-InstInstType.

7.1 Establishing Materialization Links between Instances

For materialization A—∗C, a link between an instance a of A and an instance c of C is established by inserting c into theConcreteObjects set associated with a (a → addConcreteObject(c)) and assigning a to attribute theAbstractObject of c (c→ setAbstractObject(a)).

A Boolean value is returned by addConcreteObject to report success or failure. Failure occurs when c is not an instance of C linked to a by A—∗C or if attaching c to a would violate the maximal cardinality at the A side.

Note that addConcreteObject and addAbstractObject only add link information. As discussed in Section 5, they are private methods, called by methods makeAbstractObject and makeConcreteObject of ACClass-InstType during object creation.

7.2 Deletion of Materialization Links between Instances

To break a materialization link between abstract object a and concrete object c, c is removed from theConcreteObjects set associated to a and attribute theAbstractObject associated to c is set to null.

Like addConcreteObject, removeConcreteObject returns a Boolean value to indicate success or failure. Failure occurs when c is not an instance of C linked to a by A—∗C or if deletion would violate the minimal cardinality at the A side.

Again, these methods only delete the links between instances. As discussed in Section 5, methods removeAbstractObject and removeConcreteObject are never invoked independently of method destroy of ACClass-InstType that deletes objects of application classes, since concrete objects must always be attached to an abstract object.

7.3 Querying Links between Instances

Materialization links between instances can be queried with methods getConcreteObjects and getAbstractObject.

Method getConcreteObjects returns all concrete objects of an abstract object a (the contents of theConcreteObjects attribute).

Method getAbstractObject returns the value of attribute theAbstractObject.

Method class*() extends class, supposed to be provided by the target system to return the class of a given object. Method class*() behaves like class if the target object is a root abstract object. Otherwise, class* returns the superclass of the class facet of the target object. For instance, in Fig. 18, a→ class*() and a→ class() return A, while c→ class*() returns C and c→ class() returns Cf_a.

8 Conclusion

This paper has presented an implementation for materialization, a powerful and ubiquitous generic relationship relating a class of abstract categories and a class of more concrete objects. New and powerful attribute-propagation mechanisms that generalize usual inheritance are naturally associated with materialization.

Our implementation relies upon a target system supposed to provide the following facilities: an object model with objects and classes, classification and generalization, a generic type for objects and attributes as place holder for classes and attributes to be instantiated in applications, and a metaclass concept, which plays a key role in the implementation. From the metaclass concept, we require the possibility of explicitly creating metaclasses and of attaching several classes as instances of the same user-defined metaclass. To implement materialization with a metaclass, we also assume the possibility of defining two abstract data types in the metaclass: an instance-type, for the structure and behavior of application classes involved in the relationship, and an instance-instance-type, for the instances of these classes. The target system also assumes the availability of schema update operations, namely, creation and deletion of a class as a subclass of a given class and modification of the domain of an attribute of a given class.

Under these assumptions, we built the AbstractConcreteClass metaclass as a template to capture the semantics of materialization at both the class level and the instance level. At the class level, the metaclass provides classes with the means for defining and querying the materialization links between them; it also allows them to create and delete their instances according to the semantics of materialization, whereas, at the instance level, AbstractConcreteClass provides the instances of its instances with structure and methods for establishing, deleting, and querying the materialization links between them. Furthermore, the metaclass provides for implementing attribute propagation between abstract instances and their materialized objects.

The organization of our implementation of materialization has a number of advantages. The metaclass approach avoids including the semantics of the relationship into the code of application classes, which would alter their original purpose. Also, code is not replicated every time a materialization is defined between application classes; instead, the code is written once and for all and reused through instantiations of the metaclass. The choice of an abstract target system with powerful metaclass and schema update features frees our implementation of specific decisions linked to a particular system. The implementation can thus be made simpler and more general, concentrating on essential mechanisms linked to a generic version of materialization.

In addition, our presentation clearly demonstrates the interest of extensions to existing systems to help support generic relationships. For example, the open object database system VODAK, supporting a version of metaclass, could not capture the semantics of attribute propagation associated with materialization as it provides no operation for dynamic schema evolution and no generic type for attributes.

The alternative to making materialization directly available as a generic construct is to bury its semantics as extra layers of relational-like values in simpler data structures and as additional code in the applications, for example, in the form of event-condition-action rules of relational or object-relational database systems. With such implementations, there is no way to define the generic semantics of materialization once and for all and reuse it for all specific materializations in applications. Thus, in a very real sense, information is lost and maintenance and evolution of database schemas are made more difficult because of a poorer perception of their information content.

Our work on materialization has several continuations. An interesting one deals with the propagation of methods from the abstract to the concrete class. For example, generic methods (get, set, add, remove, etc.) can be defined in the abstract class and be adapted (i.e., inherited with modifications based on the type of attribute propagation) to corresponding attributes of the concrete class. Another example deals with methods that are defined for abstract objects and that return values from concrete objects; thus, in a materialization BookTitle—∗BookCopy, describing, e.g., the inventory of a library, method borrow requests a BookTitle and returns related BookCopy objects.

We are studying the metaclass formalization and implementation of other semantic relationships for object models in various metaclass systems, in particular, VODAK and ConceptBase. They include an enriched aggregation model [50], roles [51], and ownership [52]. This will help define common metalevel specifications for relationship support and corresponding enhancements to object-oriented models.

ACKNOWLEDGMENTS

The metaclass integration of the part-of relationship [14] in VODAK provided a vigorous initial inspiration for the authors work. Additional insights about VODAK were provided by Wolfgang Klas.

REFERENCES

[1] J. Mylopoulos, "Information Modeling in the Time of the Revolution," *Information Systems,* vol. 23, nos. 3-4, pp. 127–155, 1998.

[2] A. Pirotte, E. Zimányi, and M. Dahchour, "Generic Relationships in Information Modeling," Technical Report YEROOS TR-98/09, Université catholique de Louvain, Belgium, submitted for publication, Dec. 1998.

[3] M. Dahchour, "Integrating Generic Relationships into Object Models Using Metaclasses," PhD thesis, Dept. Computing Science and Eng., Université catholique de Louvain, Belgium, Mar. 2001. Available at http://yeroos.isys.ucl.ac.be/file.pdf/PhD-01-01.pdf.

[4] A. Pirotte, E. Zimányi, D. Massart, and T. Yakusheva, "Materialization: A Powerful and Ubiquitous Abstraction Pattern," *Proc. 20th Int'l Conf. Very Large Data Bases, (VLDB '94),* J. Bocca, M. Jarke, and C. Zaniolo, eds., pp. 630–641, 1994. Available at http://yeroos.isys.ucl.ac.be/file.pdf/P-94-01.pdf.

[5] O. Yang, M. Halper, J. Geller, and Y. Perl, "The OODB Ownership Relationship," *Proc. Int'l Conf. Object-Oriented Information Systems (OOIS '94),* D. Patel, Y. Sun, and S. Patel, eds., pp. 278–291, 1994.

[6] R.J. Wieringa, W. De Jonge, and P. Spruit, "Using Dynamic Classes and Role Classes to Model Object Migration," *Theory and Practice of Object Systems,* vol. 1, no. 1, pp. 61–83, 1995.

[7] R. Motschnig-Pitrik and V.C. Storey, "Modelling of Set Membership: The Notion and the Issues," *Data & Knowledge Eng.,* vol. 16, no. 2, pp. 147–185, 1995.

[8] E. Bertino, "A View Mechanism for Object-Oriented Databases," *Proc. Third Int'l Conf. Extending Database Technology (EDBT '92),* A. Pirotte, C. Delobel, and G. Gottlob, eds., 1992.

[9] R. Motschnig-Pitrik and J. Mylopoulos, "Semantics, Features, and Applications of the Viewpoint Abstraction," *Proc. Eighth Int'l Conf. Advanced Information Systems Engineering (CAiSE '96),* P. Constantopoulos, J. Mylopoulos, and Y. Vassiliou, eds., 1996.

[10] E. Andonoff, G. Hubert, A. Parc, and G. Zurfluh, "Integrating Versions in the OMT Models," *Proc. 15th Int'l Conf. Conceptual Modeling (ER '96),* B. Thalheim, ed., pp. 472–487, 1996.

[11] H. Kilov and J. Ross, *Information Modeling: An Object-Oriented Approach.* Prentice Hall, 1994.

[12] Y. Lahlou and N. Mouaddib, "Relaxing the Instantiation Link: Towards a Content-Based Data Model for Information Retrieval," *Proc. Eighth Int'l Conf. Advanced Information Systems Eng. (CAiSE '96),* P. Constantopoulos, J. Mylopoulos, and Y. Vassiliou, eds., 1996.

[13] R. Gupta and G. Hall, "An Abstraction Mechanism for Modeling Generation," *Proc. Eighth Int'l Conf. Data Eng. (ICDE '92),* F. Golshani, ed., pp. 650–658, 1992.

[14] M. Halper, J. Geller, Y. Perl, and W. Klas, "Integrating a Part Relationship into an Open OODB System Using Metaclasses," *Proc. Third Int'l Conf. Information and Knowledge Management (CIKM '94),* N.R. Adam, B.K. Bhargava, and Y. Yesha, eds., pp. 10–17, 1994.

[15] W. Klas, K. Aberer, and E. Neuhold, "Object-Oriented Modeling for Hypermedia Systems Using the VODAK Modeling Language," *Advances in Object-Oriented Database Systems,* A. Dogaç, M.T. Özsu, A. Biliris, and T. Sellis, eds., NATO ASI Series, Springer-Verlag, 1994.

[16] M. Dahchour, A. Pirotte, and E. Zimányi, "Definition and Application of Metaclasses," *Proc. 12th Int'l Conf. Database and Expert Systems Applications (DEXA '01),* H.C. Mayr, J. Lazansky, G. Quirchmayr, and P. Vogel, eds., pp. 32–41 2001. Available at http://yeroos.isys.ucl.ac.be/file.pdf/P-01-01.pdf.

[17] D. Massart and J. Richelle, "Object-Oriented Conceptual Modeling of Databases for Macromolecular Structures," *Information Modelling and Knowledge Bases IX,* P.J. Charrel, H. Jaakkola, H. Kangassalo, and E. Kawaguchi, eds., pp. 146–159, IOS Press, 1998.

[18] Y. Tabourier, "Les Power Types ont 20 Ans," *Ingénierie des Systèmes d'Information,* vol. 5, no. 5, 1997.

[19] J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, and W. Lorensen, *Object-Oriented Modeling and Design.* Prentice Hall, 1991.

[20] R.C. Goldstein and V.C. Storey, "Materialization," *IEEE Trans. Knowledge and Data Eng.,* vol. 6, no. 5, pp. 835–842, Oct. 1994.

[21] M. Dahchour, "Formalizing Materialization Using a Metaclass Approach," *Proc. 10th Int'l Conf. Advanced Information Systems Eng. (CAiSE '98),* B. Pernici and C. Thanos, eds., pp. 401–421 June 1998. Available at http://yeroos.isys.ucl.ac.be/file.pdf/P-98-02.pdf.

[22] M. Dahchour, "Algorithms for Metaclass Implementation of Materialization," Technical Report YEROOS TR-96/07, Université catholique de Louvain, Belgium, Sept. 1996.

[23] E. Zimányi, "Implementing Materialization in Logtalk," Technical Report YEROOS TR-97/09, Laboratoire de Bases de Données, Département d'Informatique, Ecole Polytechnique Fédérale de Lausanne, Switzerland, Apr. 1997.

[24] J. Martin and J. Odell, *Object-Oriented Methods: A Foundation—UML Edition,* second ed. Prentice Hall 1998.

[25] R. Johnson and B. Woolf, "Type Object," *Pattern Languages of Program Design 3,* R. Martin, D. Riehle, and F. Buschmann, eds., Addison-Wesley, 1998.

[26] M. Fowler, *Analysis Patterns: Reusable Object Models.* Addison-Wesley, 1997.

[27] D. Hay, *Data Models Patterns: Conventions of Thought.* Dorset House Publishing, 1996.

[28] L. Al-Jadir, T. Estier, G. Falquet, and M. Léonard, "Evolution Features of the F2 OODBMS," *Proc. Fourth Int'l Conf. Database Systems for Advanced Applications (DASFAA'95),* T.W. Ling and Y. Masunaga, eds., 1995.

[29] G. Falquet, M. Léonard, and J. Sindayamaze, "F2-Concept: A Database System for Managing Classes Extensions and Intensions," *Information Modeling and Knowledge Bases V,* H. Jaakola, H. Kangassalo, T. Kitahashi, and A. Márkus, eds., pp. 243–256, IOS Press, 1994.

[30] M. Biajiz, C. Traina, and M. Vieira, "SIRIUS: Modelo de Dados Orientado a Objetos e Baseado em Abstraçoes de Dados," *Proc. Simpósio Brasileiro de Banco de Datos, XI SBBD,* pp. 338–352, 1996.

[31] P. Coad, D. North, and M. Mayfield, *Object Models: Strategies, Patterns, and Applications.* Yourdon Press, 1995.

[32] A. Pirotte and D. Massart, "La Matérialisation Pour Réconcilier Deux Descriptions des Taxinomies," *Proc. 7è Rencontres de la Société Française de Classification,* Sept. 1999. Available at http://yeroos.isys.ucl.ac.be/file.pdf/P-99-02.pdf.

[33] A. Pirotte and D. Massart, "Integrating Two Descriptions of Taxonomies with Materialization," Technical Report YEROOS TR-00/01, Université Catholique de Louvain, Belgium, Jan. 2000. Submitted for publication.

[34] J. Mylopoulos, P. Bernstein, and H. Wong, "A Language Facility for Designing Interactive, Database-Intensive Applications," *ACM Trans. Database Systems*, vol. 5, no. 2, 1980.

[35] M.L. Brodie and D. Ridjanovic, "On the Design and Specification of Database Transactions," *On Conceptual Modelling*, M.L. Brodie, J. Mylopoulos, and J.W. Schmidt, eds., Springer-Verlag 1984.

[36] M. Jarke, R. Gallersdörfer, M.A. Jeusfeld, and M. Staudt, "ConceptBase : A Deductive Object Base for Meta Data Management," *J. Intelligent Information Systems*, vol. 4, no. 2, pp. 167–192, 1995.

[37] R. Fikes and J. Kehler, "The Role of Frame-Based Representation in Reasoning," *Comm. ACM*, vol. 28, no. 9, Sept. 1985.

[38] J. Mylopoulos, A. Borgida, M. Jarke, and M. Koubarakis, "Telos: Representing Knowledge about Informations Systems," *ACM Trans. Office Information Systems*, vol. 8, no. 4, pp. 325–362, 1990.

[39] G. Kiczales, J. des Rivières, and D. Bobrow, *The Art of the Metaobject Protocol.* MIT Press, 1991.

[40] P. Moura, "Logtalk 2.5 User Manual," Center for Informatics and Systems, Univ. of Coimbra, Portugal, Dec. 1999.

[41] T. Ledoux and P. Cointe, "Explicit Metaclasses as a Tool for Improving the Design of Class Libraries," *Proc. Int'l Symp. Object Technologies for Advanced Software (ISOTAS'96)*, pp. 38–55, 1996.

[42] A. Goldberg and D. Robson, *Smalltalk-80: The Language and Its Implementation.* Addison-Wesley, 1983.

[43] P. Butterworth, A. Ottis, and J. Stein, "The Gemstone Database Management System," *Comm. ACM*, vol. 34, no. 10, pp. 64–77, 1991.

[44] W. Klas and M. Schrefl, "Metaclasses and Their Application," *Lecture Notes in Computer Science 943*, Springer-Verlag, 1995.

[45] M. Kolp, "A Metaobject Protocol for Reifying Semantic Relationships into Open Systems," *Proc. Fourth Doctoral Consortium Ninth Int'l Conf. Advanced Information Systems Eng. (CAiSE '97)*, pp. 89–100, June 1997. Available at http://yeroos.isys.ucl.ac.be/file.pdf/P-97-03.pdf.

[46] M. Kolp, "A Metaobject Protocol for Integrating Full-Fledged Relationships into Reflective Systems," PhD thesis, INFODOC, Université Libre de Bruxelles, Belgium, Oct. 1999. Available at http://yeroos.isys.ucl.ac.be/file.pdf/PhD-99-01.pdf.

[47] C. Marcos, M. Campo, and A. Pirotte, "Reifying Design Patterns as Metalevel Constructs," *Proc. Second Argentine Symp. Object Orientation (ASOO '98)*, Aug. 1998. Also published in the *Electronic J. SADIO (Argentine Soc. Informatics and Operation Research)*, vol. 2, no. 1, pp. 17–29, Available at http://yeroos.isys.ucl.ac.be/file.pdf/P-98-10.pdf, 1999.

[48] M.A. Jeusfeld, M. Jarke, M. Staudt, C. Quix, and T. List, "Application Experience with a Repository System for Information Systems Development," technical report, Tilburg Univ., The Netherlands, Feb. 1999.

[49] M. Halper, J. Geller, and Y. Perl, "An OODB Part-Whole Model: Semantics, Notation, and Implementation," *Data & Knowledge Eng.*, vol. 27, no. 1, pp. 59–95, May 1998.

[50] M. Dahchour, M. Kolp, and A. Pirotte, "Metaclass Implementation of a Part-Relationship Model," Technical Report YEROOS TR-98/10, Université catholique de Louvain, Belgium, Dec. 1998. Submitted for publication.

[51] M. Dahchour, A. Pirotte, and E. Zimányi, "A Generic Role Model for Dynamic Objects," *Proc. 14th Int'l Conf. Advanced Information Systems Eng., (CAiSE '02)*, T. Ozsu, J. Mylopoulos, and C. Woo, eds., May 2002. Available at http://yeroos.isys.ucl.ac.be/file.pdf/P-02-01.pdf.

[52] M. Halper, Y. Perl, O. Yang, and J. Geller, "Modeling Business Applications with the OODB Ownership Relationship," *Proc. Third Int'l Conf. AI Applications on Wall Street*, R.S. Freedman, ed., pp. 2–10, June 1995.

Mohamed Dahchour received the BSc degree (1993) in computer science from the Université Paul Sabatier, Toulouse, France. He received the MSc degree (1994) and the doctorate degree (2001) in computer science from the Faculté des Sciences Appliquées, Université catholique de Louvain. He is a teaching assistant at the IAG School of Management at the Université catholique de Louvain in Louvain-la-Neuve. His research interests include conceptual modeling, semantic relationships, metamodeling techniques, and object-oriented databases.

Alain Pirotte is a professor of computer science and information management at the Université catholique de Louvain in Louvain-la-Neuve and at the Université Libre de Bruxelles, Belgium. Earlier, he was a researcher in industry for more than 20 years. His scientific interests include database management, information modeling, object technology, and the methodology for software development. He is a member of the IEEE Computer Society.

Esteban Zimányi started his studies at the Universidad Autónoma de Centro América, Costa Rica. He received the BSc degree (1988) and the doctorate degree (1992) in computer science from the Faculté des Sciences at the Université Libre de Bruxelles (ULB). Currently, he is a professor in the Engineering Department at ULB. From 1992 to 1998, he was a lecturer in the Department of Information and Documentation of the ULB. During 1997, he was a visiting researcher at the Database Laboratory of the Swiss Federal Institute of Technology in Lausanne, Switzerland. His current research interests include conceptual modeling, semantic relationships, geographic information systems, and temporal databases.