Materialization: a powerful and ubiquitous abstraction pattern

Alain Pirotte, * Esteban Zimányi, David Massart, † Tatiana Yakusheva ‡

Abstract

Materialization is a useful abstraction pattern that can be identified in many application settings. Intuitively, materialization is the relationship between a class of categories (e.g., models of cars) and a class of more concrete objects (e.g., individual cars). This paper gives a quasi-formal semantic definition of materialization in terms of the usual is-a and is-of abstractions, and of a class/metaclass correspondence. New and powerful inheritance mechanisms are associated with materialization. Examples, properties, and extensions of materialization are also presented. Providing materialization as an abstraction mechanism for conceptual modeling enhances expressiveness by a controled introduction of classification at the application level.

Keywords: conceptual modeling, object oriented model, classification.

1 Introduction

Conceptual modeling is the activity of formalizing some aspects of the physical and social world around us for purposes of understanding and communication. Advances in conceptual modeling involve narrowing the gap between concepts in the real world and their representation in conceptual models by identifying new powerful modeling primitives that allow a more accurate and intuitive description of real world concepts.

Object-oriented analysis strives to understand and model application domains with object-oriented concepts from the point of view of users and domain experts, without worrying about design and implementation. Semantically rich languages can substantially ease the task of analysts. A realistic way to extend conceptual languages is to enrich them with high level domain-oriented patterns [Coa92]. Another advantage of high level patterns is that they can be reused by instantiation in related application domains. This paper presents and formally defines one such extension.

Figure 1 introduces an example of materialization that we will expand later in the paper. It relates two classes, *Car* and *Car_model*, with attributes *serial#* and *manuf_date*, and *name*, *sticker_price*, and *#doors*, respectively.

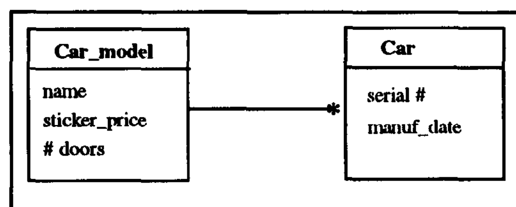


Figure 1: A first example of materialization.

Car_model represents information typically displayed in the catalog of car dealers, whereas *Car* represents information about individual cars owned by people. An instance of class *Car_model* describes a specific model (e.g., with *name* = `Fiat_Retro`) and defines attributes common to all cars of that model.

Materialization is a special relationship between two

*University of Louvain, IAG, 1 place des Doyens, 1348 Louvain-la-Neuve, Belgium, e-mail: pirotte@info.ucl.ac.be

†University of Brussels, 50 Av. F. Roosevelt, C.P. 197, 1050 Brussels, Belgium, e-mail: {ezimanyi,dmassart}@ulb.ac.be

‡University of Louvain, IAG, 1 place des Doyens, 1348 Louvain-la-Neuve, Belgium, e-mail: yakusheva@qant.ucl.ac.be.

T. Yakusheva is supported by the FDS program (Fonds de Développement Scientifique) at the University of Louvain.

Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the VLDB copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Very Large Data Base Endowment. To copy otherwise, or to republish, requires a fee and/or special permission from the Endowment.

Proceedings of the 20th VLDB Conference
Santiago, Chile, 1994

classes, where one (here, *Car_model*) is more abstract than the other (here, *Car*). Following [GS94], we note materialization as a straight line with a star * on the side of the more concrete class.

Defining materialization consists in precisely characterizing the properties of the relationship. Thus, materialization has a flavor of is-a generalization (we are interested in a class of cars and in its subclasses for specific models) and also a flavor of is-of classification (a specific car model, e.g., `Fiat_Retro`, can be modeled as an instance of a class of car models). But the desired semantics is not adequately captured by either of those abstraction mechanisms alone.

Every concrete car (e.g., `Nico`'s car) has exactly one model (e.g., `Fiat_Retro`), while there can be any number of cars of a given model. Thus materialization has cardinality (1,1) on the side of the more concrete class and cardinality (0,n) on the side of the more abstract class. Concrete cars somehow inherit some of the properties of their model: we will thus have to characterize the inheritance properties of materialization.

The rest of the paper is structured as follows. The basis of our data model is a fairly typical object-oriented model, that we briefly describe in Section 2, insisting on the most important assumptions. There we also introduce a partial order to model the relative abstractness/concreteness character of classes. Section 3 presents a general classification pattern that combines an instance of both is-a and is-of relationships. Section 4 is the core of the paper with a nearly formal definition of materialization. We devote Section 5 to the more delicate question of inheritance mechanisms for materialization. New inheritance mechanisms are characterized, which require the basic semantic definition to be extended. Section 6 presents further examples, while Section 7 is devoted to combinations and extensions of materialization. Previous work on materialization is discussed in Section 8, while Section 9 is a conclusion with suggestions for subsequent work.

For lack of space, this paper contains no strictly formal definitions. However, Sections 4 and 5 give all the necessary ingredients for a formal definition, which is discussed in a companion paper [PZ94].

2 Our object model

As a framework for our study of materialization, we assume a typical modern object model, with object classes, relationships, abstraction mechanisms (classification, generalization, aggregation) (see, e.g., [ABD+89, Dit93, Kim90, LAC+93, ZM90]). We also need a class/metaclass correspondence and we introduce a partial order to model the abstractness/concreteness dimension.

Classification (is-of, instance-of, is-member-of). We are well aware that there is a whole range of possible variations for defining classification and the class/metaclass correspondence (see for example the careful discussion in [MPM92]). For clarity, we assume a simple version of object model, knowing that our discussion of materialization may have to be adapted, should another version be adopted.

Classification associates a class with a set of objects with the same properties. Thus, we assume for the moment that all instances of a class are isomorphic or structurally equivalent. The class is the place where properties common to all objects of the class are defined. These properties can be classified into structural (attributes) and behavioral (methods) depending on whether they describe the structure of objects or their behavior. We relax structural equivalence in Section 5.

Class/metaclass correspondence. Viewed as an object, a class is related to another class, its *metaclass*, by an is-of link. Metaclasses are explicitly supported by several semantic models (e.g., TAXIS [MBW80], SHM [BR84]), object models (e.g., VODAK [KAN93], ADAM [PD91], OSCAR [GH93]), knowledge representation languages (e.g., LOOPS [BS83], KEE [FK85], Proteus [Rus89], Telos [MBJK90, JJ91, JEG+94]) and programming languages (e.g. Smalltalk-80 [GR83], ObjVlisp [Coi87], CLOS [Kee89]).

The common use of classification in database management is for modeling the basic type/instance dichotomy: classification is the relationship, essentially implicit as an abstraction mechanism, between the schema and the extension of the database. It is uncommon to mix classes and instances in database conceptual modeling. But when the class/metaclass correspondence is made available as an abstraction mechanism, then classification becomes explicit within the conceptual schema.

A class and the associated object instance of its metaclass are conveniently modeled as a *two-faceted* construct [Van90]. We will sometimes write Class(C) and Object(C), respectively, to specifically refer to each facet. Attributes of the metaclass instantiate as values for the object facet. It is convenient to consider that these attribute values transfer as attribute types of the class facet, with the provision that they are constant attributes, i.e., that their value is the same for all instances of the class facet. Consistently with the literature (see e.g., [MPM92]), we call them *class attributes* of the class facet. The other attributes of the class facet, that are instantiated for each instance of the class, are referred to as *instance attributes*. We generalize this mechanism in Section 5. The relationship between the class facet and the object facet is abstraction: all the attributes of the object are mean-

ingful for the class, whereas the instance attributes are specific to the class and its instances.

Surprisingly little attention has been devoted so far to the semantics of such two-faceted constructs in database management. We will see that they are central to the semantics of materialization.

Relationships are included in our model, as they are in most modern object-oriented models. An instance of a relationship corresponds to exactly one instance of each of the participating classes, and an object may participate in any number of instances of a relationship. A relationship can also be viewed as a class and thus have attributes of its own. A relationship class may participate in other abstractions, like generalization, or in another relationship.

Generalizations (or inclusion, subset, is-a) are special relationships involving two (or more) classes. If X is-a Y, then the set of instances of X is included in the set of instances of Y. X is the subclass of the generalization and Y is the superclass. As usual, generalization can be total or partial, and exclusive or overlapping. Generalization realizes the inheritance property: abstractions defined for the superclass are inherited by the subclasses.

Aggregation (or part-whole, a-part-of) defines an aggregate (or composite) class in terms of constituent (or component) classes and can be viewed as a class of its own.

Abstractness: we define the abstract/concrete aspect of object classes as a relative property, rather than an absolute one as is usually done (see e.g., [GS94]). Specifically, the more or less abstract or concrete character of object classes is described by a user-supplied partial order among classes. Thus, given two classes A and C, either A is more abstract than C, or C is more abstract than A, or nothing is said about the relative abstractness or concreteness of A and C.

As usual when defining a conceptual language, we are only interested in the formal (or formalizable) semantics, i.e., the semantics that can be formally related to the constructs of the basic object model, that we distinguish from the intuitive unformalized semantics of concepts in the outside world. The latter is sometimes referred to as the *particular semantics* of the application domain. The relationship between concepts in the conceptual schema and the corresponding particular semantics is sometimes called the *denotation* of the formal constructs in the application domain.

To summarize, two ingredients are new in this section: our use of the term *two-faceted construct* to refer to the composite construct made of an object and its associated class, and the introduction of a *partial order* to model the more or less abstract character of object classes. The discussion of *inheritance*, for which we propose new mechanisms, is postponed to Section 5. The rest of this section borrowed from fairly classical object models to characterize the basic framework that we use in the rest of the paper.

Notations are important, as we deal with concepts that can play different roles. We denote classes as square boxes, instance objects as oval boxes, classification links with grey lines, and is-a generalization links with solid black lines. To picture their double role, we draw a two-faceted construct as a square box adjacent to an oval box. Class names are written in italics with their first letter in upper-case, class attributes in lower-case italics, constants in typewriter font. In the figures, attribute types are not explicitly indicated, except when they exhibit an essential aspect of the semantics of materialization.

3 A taxonomic pattern

This section presents, by means of an example, a taxonomic pattern that combines an instance of generalization and of classification. This pattern illustrates the class/metaclass correspondence and the use of the associated two-faceted constructs in conceptual modeling. It prepares our definition of the semantics of materialization in the next section.

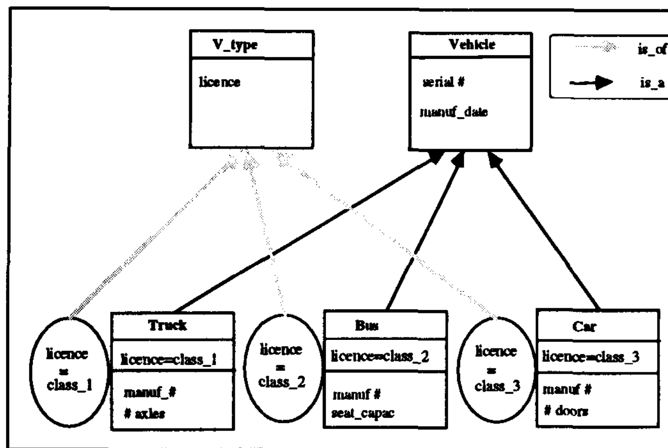


Figure 2: A general classification pattern.

The example models a situation where there is an interest in a class/subclass generalization and at the same time about the categories of objects in the subclasses, along a meta dimension. Categories of objects are themselves objects about which useful information is described.

Consider a class of land vehicles (*Vehicle*), which

generalize classes of trucks (*Truck*), buses (*Bus*), and cars (*Car*). We are also interested in the classification itself of the types of vehicles that we model through a class of vehicle types (*Vtype*).

Figure 2 shows the corresponding conceptual schema, where two-faceted constructs emphasize that *Truck*, *Bus*, and *Car* are at the same time instances of the metaclass *Vtype* and subclasses of the class *Vehicle*.

For example, *Truck* as a class defines attribute types, e.g., *manuf* or *#axles* for its instances. Such instance attributes do not concern *Truck* viewed as an instance of class *Vtype*. As an object, *Truck* possesses a value for the instance attributes of the metaclass *Vtype* (e.g., *licence* = `class_1`). Such class attributes are inherited by all the instances of class *Truck*. Similar taxonomies, sometimes with many classification levels, are commonplace, for example in the natural sciences (e.g., organisms in biology).

Most often, in the database literature, the meta dimension is treated as a step away from the application domain towards the software environment that describes or manipulates the application domain (see, e.g., [GH93, KAN93]). Here, the classification abstraction is used to model information that is useful in the application domain itself. The metalevel is merged with the so-called object level to define the language available for modeling the application domain. This merging of levels to produce a more powerful language is analog to what was called "reflection" in programming languages, namely the merging, into an extended language, of LISP or Prolog and their metalevels interpreters [BH88, MN88]. We will see that such a two-faceted description of something which is a single concept in the real world is the essence of materialization.

4 Semantics of basic materialization

This section presents the basic semantics of materialization. It is generalised in Section 5 when new inheritance mechanisms are defined.

Informal definition

Figure 3 sketches the semantic definition of the example in Figure 1. Its structure is similar to that of Figure 2. The essential difference is that the whole graph of Figure 2 is part of the conceptual schema, while, for materialization, of course only the two classes related by materialization appear in the conceptual schema. The two-faceted constructs make explicit the semantics of materialization but they do not belong to the conceptual schema. For clarity, we enclose the semantic parts in a shaded box; only the unshaded parts appear in the conceptual schema.

Figure 3 exhibits two instances of the more abstract class *Car_model*, with *name* `Fiat_Retro` and `Wild_2CV`, respectively. They are the object facets of two-faceted constructs whose class facets are subclasses of the more concrete class *Car*. Thus, for example, a two-faceted construct reconciles the views of `Fiat_Retro` as both an object of class *Car_model* and a class *Fiat_Retro_Cars* of concrete cars of that model.

We have already pointed out that the relationship between the class and the object aspects in a two-faceted construct is abstraction: there is less information in the `Fiat_Retro` object than in the *Fiat_Retro_Cars* class associated to it. Now, considering the reverse "concretion" process, we see that the attribute information added to every *Car_model* object to produce a class is the same: it consists of instance attributes (*serial#* and *manuf_date* in the example), in the terminology of Section 2, that each class of cars of a particular model inherits from class *Car* through an is-a generalization.

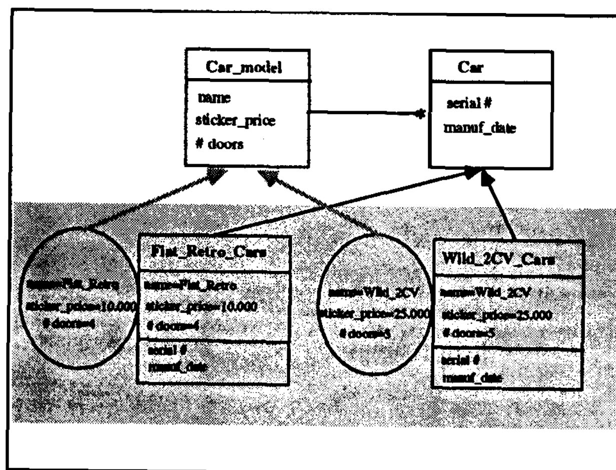


Figure 3: Semantics of the example in Figure 1.

Thus, to summarize, the classes *Fiat_Retro_Cars* and *Wild_2CV_Cars* of the two-faceted constructs have as attributes: (1) the class attributes from the corresponding instance of the more abstract class *Car_model*, and (2) the instance attributes inherited from the more concrete class *Car*.

Nearly formal definition

We now summarize, in a precise way, the necessary elements for a formal definition of materialization, which is discussed in [PZ94].

Materialization is a binary relationship R between two classes A and C where A is more abstract than C and the cardinality of R is (1,1) on the C-side and (0,n) on the A-side.

The semantics of the materialization of A into C is expressed as a composition of the class/metaclass correspondence, and the is-a generalization and is-of

classification abstractions, as sketched in Figure 4.

The semantics is expressed with a collection of two-faceted constructs C_i. Following the notations of Section 2 for two-faceted constructs, for each C_i, class Class(C_i) is a subclass of C and Object(C_i) is an object instance of A.

Thus, materialization induces a partition of the population of class C into subclasses Class(C_i) along an is-a dimension. This partitioning of class C is governed by the is-of classification link: there is a class Class(C_i) of the partition for each object instance Object(C_i) of class A.

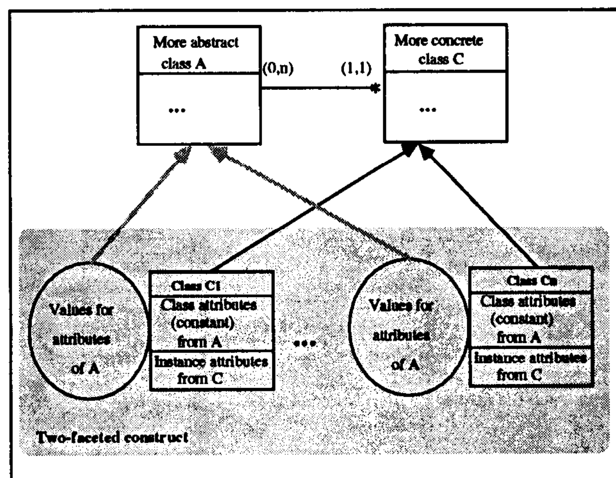


Figure 4: Semantics of materialization.

As an instance of A, each object Object(C_i) has a value for each instance attribute of class A. These attribute values are transferred as class attributes to the associated Class(C_i), that is, as attributes with the same value for each instance of Class(C_i). The instance attributes of class(C_i) are inherited from C through the is-a generalization link. Thus they are the same for all C_i's. There are no other attributes for Class(C_i).

To summarize, the semantics of the materialization of a more abstract class into a more concrete class is a partition (i.e., a total and exclusive generalization) of the more concrete class governed by the instances of the more abstract class.

5 Inheritance

We agree in general with e.g. [Bra83, MPM92] that, beyond the common intuition, the fine interpretation of inheritance is far from unequivocal and that inheritance is, in the first place, a convenient implementation mechanism. Still, in all examples of materialization, many attributes of the more abstract class are naturally applicable to the more concrete class and we find convenient to present the discussion in terms of inheritance mechanisms.

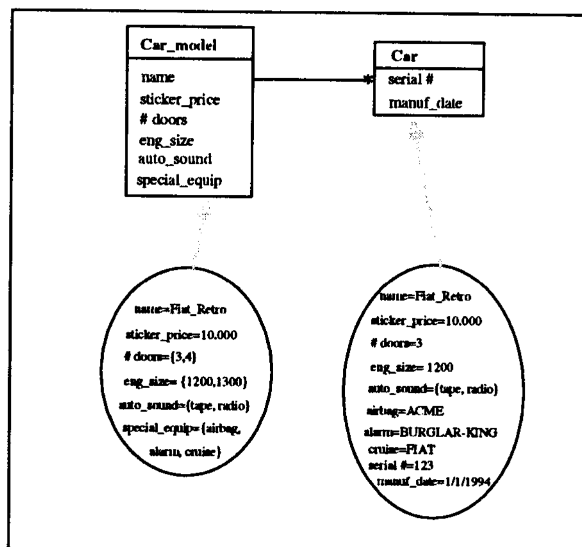


Figure 5: Example of Figure 1 with more attributes.

Figure 5 expands the example of Figure 1 by adding multivalued attributes ***eng_size, auto_sound*** and ***special_equip*** to class *Car_model.* Figure 5 also shows an instance of each class. The semantics of the example is sketched in Figure 6, which similarly expands Figure 3, with two examples of two-faceted constructs.

According to our definition of materialization, each instance of *Car_model* is the object facet of a two-faceted construct, whose class facet is a class of the partition of class *Car* governed by the corresponding instance of class *Car_model.* We distinguish three mechanisms for inheritance from the more abstract class to the more concrete one, that we describe as correspondences from an attribute value of the instance facet to a class attribute of the class facet of a two-faceted construct:

- **Type 1.** The value of an attribute of the object facet is inherited as a class attribute of the class facet. An example is the monovalued attribute *name.* This mechanism also applies to multivalued attributes. Thus a multivalued attribute *standard_equipment* of *Car_model* could list the standard equipment of each car model, that is exactly the equipment that comes with each physical car of the model.
- **Type 2.** The value of a multivalued attribute of the object facet becomes the ***domain*** of an attribute, monovalued or multivalued, of the class facet.

An example of the monovalued case is exhibited by the multivalued attribute *eng_size* of *Car_model*. Its value {`1200,1300`} for the `Fiat_Retro` object facet means that physical cars of the `Fiat_Retro` model can come with *eng_size* = `1200` or with *eng_size* = `1300`. Thus the associated class *Fiat_retro_cars* has a monovalued attribute *eng_size* with {`1200,1300`} as its type.

An example of the multivalued case is exhibited by the multivalued attribute *auto_sound* of *Car_model*. Its value {`tape,radio`} for the `Fiat_Retro` object facet means that physical cars of the `Fiat_Retro` model can come with `tape`, or `radio`, or both, or nothing at all as *auto_sound*. Thus the associated class *Fiat_retro_cars* has a multivalued attribute *auto_sound* with {`tape,radio`} as its type.

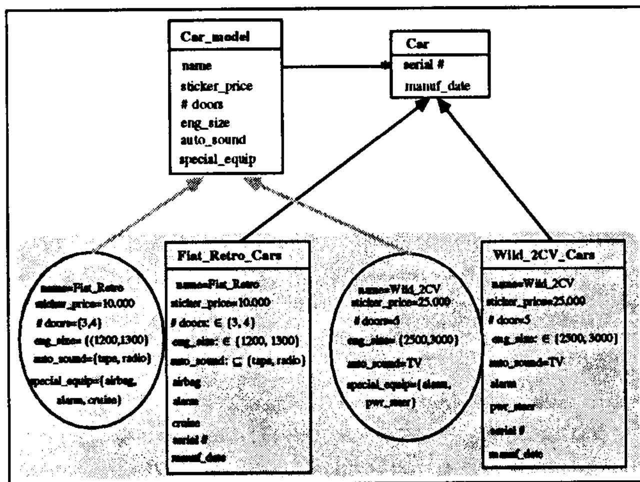


Figure 6: Semantics of the example in Figure 5.

- **Type 3.** The *value* of a monovalued (resp., multivalued) attribute of the object facet determines one (resp., several) *new attributes* of the class facet.

  An example of the multivalued case is exhibited by the multivalued attribute *special_equip* of *Car_model*. Its value {`Airbag,Alarm,Cruise`} for the `Fiat_Retro` object facet means that physical cars of the `Fiat_Retro` model come with three pieces of special equipment: an `Airbag`, an `Alarm`, and a `Cruise_ctrl`. In addition, for each piece of equipement, various models are available. Thus the associated class *Fiat_retro_cars* has three monovalued attributes *airbag, alarm*, and *cruise_ctrl*, whose value is the make of the corresponding piece of equipment.

  Another example of type 3 inheritance is developed in Section 7.1.

With the type 3 mechanism, the class facets are no longer structurally equivalent. While class *Fiat_retro_cars* has attributes *airbag, alarm*, and *cruise_ctrl*, class *Wild_2CV_cars* has *alarm* and *pwr_steer* as corresponding attributes.

In many examples of materialization, it seems that inheritance from the more abstract class to the more concrete class could be strict, with the appropriate particular semantics. For example, each *Car* object somehow carries or refers to, albeit sometimes indirectly, the properties of the *Car_model* object that it materializes.

As an example of less plausible inheritance, consider *first_year_manuf* and *total#_sold* as new attributes for class *Car_model*, with obvious particular semantics. At first glance, it does not seem that they should be treated as attributes of *Car* objects. The latter could also be defined as a derived attribute and its value for a particular *Car_model* be computed from the materialized *Car* objects of the associated *Car_model*. But it is also conceivable that both attributes be inherited by *Car* objects, and treated as, arguably remote, properties of individual cars (with the meaning of first production date for the model of a car and total number of cars sold for the model of that car).

Our feeling about the strictness of inheritance is that as much flexibility as possible should be left to the analysts modeling an application domain and that they should have the option of defining non-strict data attributes if that fits their perception of the domain.

On the contrary, we have found no example of methods that we would like the more concrete class of a materialization to inherit from the more abstract class.

Class *Car_model* could have, for example, an *add_eng_size* method that makes available a new value of *eng_size*, a type 2 attribute, for cars of a particular model. Applied to the `Fiat_Retro` object of class *Car_model*, the operation should change the definition of class *Fiat_retro_cars* so that the new type of its *eng_size* attribute include the new engine size. Thus the *eng_size* class attribute that class *Car* inherits in the materialization has changed its type in the operation.

Operations on type 3 attributes of the more abstract class cause still more drastic changes to the attribute structure of the more concrete class. Thus, with type 2 and type 3 inheritance mechanisms, appealing as they look, not only do we have to give up the the structural equivalence for instances of the more concrete class, but we also open the door to a highly dynamic conceptual schema.

6 More examples of materialization

Materialization provides a new degree of freedom for the analyst building conceptual schemas modeling a part of the real world.

For a class C to materialize a class A, C must be more concrete than A in the partial order that expresses abstractness, and the cardinalities of the relationship between A and C must be (1,1) on the side of C and (0,n) on the side of A. Note that the (0,n) cardinality could be made stricter by the real world semantics being modeled.

Once materialization is made available in the schema, examples are ubiquitous. The following gives only a short list:

- Consider the modeling of air traveling. It could 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, days of the week, period of the year, etc.), itself materialized as a class of flights for specific calendar days (with a date, an aircraft, a crew, etc.). Another analysis of the concept of trip is shown in Figure 9 (b). This example if typical of the frequent lack of specific words in natural languages to refer precisely to the various levels of materialized concepts. Often, the same word is used at various levels and ambiguity is avoided by various clues in the linguistic or pragmatic context of the communication.

- News items are materialized as articles for a particular edition of a newspaper, which in turn materialize as physical copies of the newspaper.

- Similarly, stories materialize as book titles, which materialize as book copies. Stories can also materialize as theater plays, which themselves materialize as performances. A refinement of that example is discussed in Section 7.1. Movies are another materialization of stories, and they can in turn materialize as videotapes.

- In our car example, the *Car_model* class can materialize as catalogs for car dealers and as videos presenting the models.

Section 7.1 shows that materialization is transitive, which suggests other instances of materialization from existing instances.

Conversely, we have found no examples where we could rule out materialization between two classes that satisfy the partial order. Specific choices of particular semantics always enable to discover scenarios in the application domain in which materialization is meaningful when the partial order is satisfied. Thus it appears that a necessary and sufficient condition for materialization to be possible between two classes is that they satisfy the partial order for abstractness. We do not rule out tightening this condition through a finer analysis, by detecting pieces of particular semantics that are sufficiently general to be transformed into formalizable semantics (like we did with the partial order that models abstractness/concreteness).

7 Extensions

Thus far, we have discussed materialization as a relationship between two classes. This section extends materialization in several directions: first, the composition of materializations in cascade illustrates the transitivity of materialization; then materialization is extended to relationships, aggregation, and constraints. Only the general ideas are given here by means of examples. Elaboration will be presented in forthcoming papers.

7.1 Composition of materializations

This section considers, by means of an example, cascades of materializations, where the more concrete class of a materialization is also the more abstract class of another materialization, and so on.

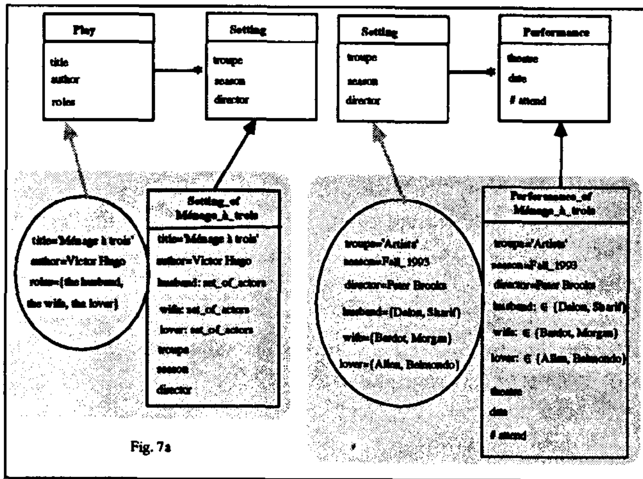
The example, shown in Figure 7, deals with theatre *Plays* written by an *author*, with a *title* and a set of main *roles*. An instance of *Play*, say, `Ménage_à_trois` by `Victor_Hugo`, has three main roles that are represented as a value `husband,wife,lover` of the multivalued attribute *role*.

Plays materialize as *Settings*, that embody the production decisions for a theatrical *season*: a *troupe*, a *director*, a set of actors for each *role* of each play at the repertoire for that season. The semantics of materialization partitions *Setting* in as many classes as there are instances of *Play*.

The multivalued *roles* attribute of *Play* is another example of type 3 inheritance. Thus the class facet of the two-faceted construct of `Ménage_à_trois` has three multivalued attributes, *husband*, *wife*, and *lover*, corresponding to its main roles. An instance of *Setting_of_Ménage_à_trois* decides for example that `Delon` and `Sharif` will be available as *husband* during the `Fall_1993` season.

Settings materialize as *Performances*, at a particular *date*, such that each *role* of *Play* is assigned to a specific actor for each *Performance*. Thus, *Performance* is partitioned in as many classes as there are instances of *Setting*. In an instance of *Performance*, for example, on 12/22/93 at the `Parc` theatre, the *husband* was `Delon`.

Figure 7a shows both materializations and sketches their semantics, by displaying one two-faceted construct for each one. Figure 7b an instance of each of the three classes involved, will all their attribute values.



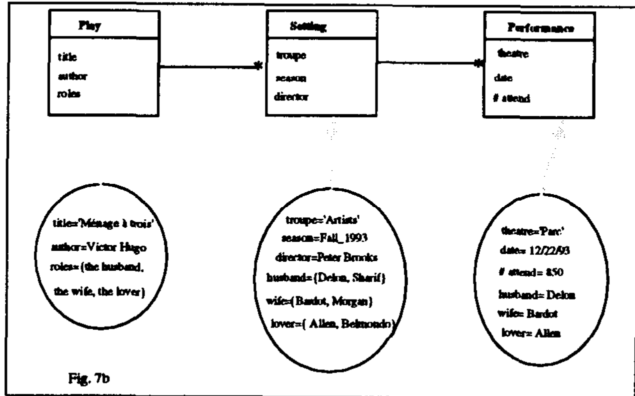


Figure 7: Composition of materializations.

This example shows that cascading materializations permits a fine analysis of the application domain. In the class *Setting_of_Ménage_à_trois*, the three new attributes correspond to the value of the *roles* attribute in the associated instance of *Play*. In the class *Performance_of_Ménage_à_trois*, the domain of attribute *husband* (and, similarly, *wife* and *lover*) denotes the set of actors available for playing the *husband* role in a particular theatrical season, represented as an instance of *Setting_of_Ménage_à_trois*.

Transitivity of materialization is a trivial property at the type level. If C materializes B and B materializes A, then C can also materialize A, since A is more abstract than C by transitivity of the partial order. But transitivity at the instance level does not necessarily hold: its presence or absence depends on the particular semantics of the application domain. For example, stories can materialize as theatre plays (e.g., the Antigone story written by Sophocles materializes as the play written by Anouilh), which can in turn materialize as physical books. Stories also materialize as books, but the book of Anouilh's play is different from that of Sophocles' story.

Even when transitivity of materialization holds at the instance level, its semantics is not a simple combination of the intermediate materializations. In our example above, *Play* can directly materialize as *Performance* without the intervening *Setting* and things can be such that transitivity holds at the instance level. The modeling is coarser and, not surprisingly, information is lost in the shortcut: it is no longer possible to model that a set of actors are available for the same role during a particular season. The mechanics of these combinations is further investigated in [PZ94].

7.2 Materialization of relationships

A relationship can be seen as an object (sometimes called *associative object*) with its own properties (see, e.g., [RBP+91]). Therefore, a relationship can participate as an object class in other abstractions, for example materialization. Figure 8 shows a relationship *Course* associating object classes *Teacher* and *Subject*, and a materialization *Course_Offer* of *Course*, with obvious meanings. The attributes of *Course* are monovalued and are inherited by *Course_Offer* through a type-1 mechanism (see Section 5).

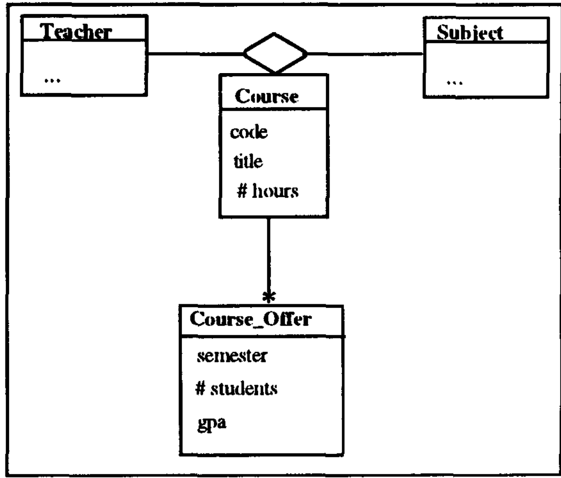


Figure 8: Materialization of relationships.

7.3 Materialization of aggregation

Consider now materialization of aggregation, as defined in, e.g., [MP93]. When an object in an aggregation hierarchy materializes into a more concrete object, the aggregation hierarchy also materializes into a more

concrete hierarchy with the same structure as that of the abstract hierarchy. A similar behavior is called homomorphism in [RBP+91].

As an example, Figure 9(a), inspired from [RBP+91], models a catalog for car parts, where a catalog part may contain other catalog parts. Each catalog part is identified by a part number, while each manufactured part has its own serial number. Like the catalog parts, the physical parts are composed of subparts. The part explosion tree has the same form for both the catalog parts and the physical parts.

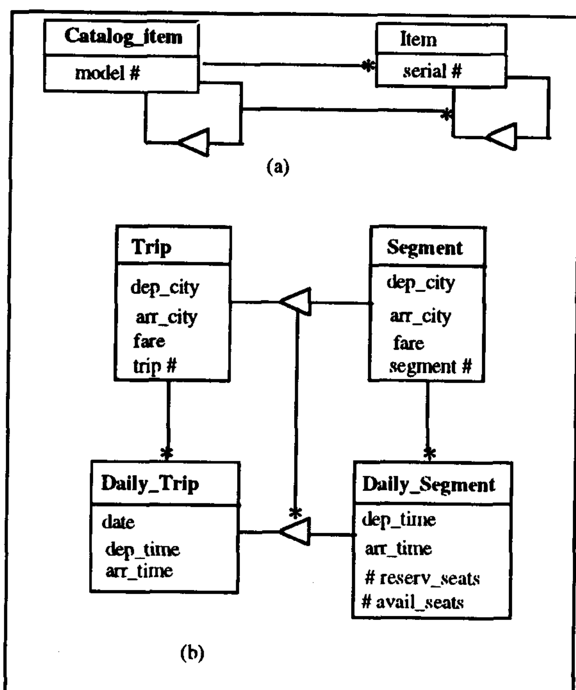


Figure 9: Materialization of composition hierarchies.

Figure 9(b), inspired from [BCN92], analyzes a concept of trip. A trip is composed of several segments. Trips and segments describe the fare information. Trips and segments materialize, respectively, as daily trips and daily segments, that contain the schedule information. The composition hierarchy of a trip in segments is materialized as a hierarchy with the same structure for the associated daily trips.

7.4 Incomplete information and materialization of constraints

The following example is from [Bee93]. A travel agent sells travel plans featuring many choices and options about flights, hotels, side trips, and so on. Eventually, travel agent and customer agree on a specific plan, with no option left open. In the meantime, travel plans exist in an incomplete state, where some choices have been made, others are still open, still others are tentative (e.g., awaiting confirmation). Also, some choices may have to be undone, because requests can be denied and because customers change their mind.

We describe plans at various stages with a generic class *Generic_plan* that materializes for example as *Aegean_cruise* or *Across_USA_tour*, which in turn materialize as several classes of *Customer_trip*, where all choices have been made. Restrictions and constraints may apply to *Generic_plan* and to the more concrete plans, and they have to be satisfied by the corresponding *Customer_trip*. Thus, materialization of constraints parallels the role of constraints in traditional databases: they are formulated in the database schema and they must be satisfied by the database instance. Here, they appear as constraints in the more abstract classes and, having fulfilled their role as constraints, they disappear as such in the more concrete classes, whose instances satisfy the constraints and for which the constraints themselves have become in a sense irrelevant.

A similar materialization of constraints is typical of engineering application domains like manufacturing, design, planning, or scheduling. Guidelines and company procedures appear as explicit constraints that the end products must satisfy. Such procedural rules may take the form of *templates* for design, with, for example, a hierarchical part/subpart structure in which subparts still to be chosen can be prescribed to be mandatory or optional, single or multiple, etc. Such templates can be represented as AND/OR trees, where OR-nodes represent choices still to be made in the design process. The designer retrieves a template for the part to be designed and records design decisions according to the prescriptions and rules of the template. Designs have to coexist at various stages of completion in the design database.

Templates can be represented as one or several more abstract classes, possibly with more concrete materialized templates where some design decisions have been made. Completed designs are modeled as one or several more concrete classes. Several levels data manipulation must be supported. For example, "do equipments A and B always have a part in common?" is a query at a more abstract level than "if we choose part #333 as a subpart of part A, do all possible designs cost more than $100 ?", which is at the design level.

Several questions are raised by this kind of applications. Some were studied in [INV91a, INV91b], but no differentiation is made there between abstract and concrete classes. Another problem is that the aggregation abstraction (as studied, e.g., in [MP93]) is not powerful enough to represent a family of alternative part/subpart hierarchies. At the more concrete design level, the main problem is to cope with incomplete information, for which satisfactory solutions are still lacking [ZP92]. In particular, very little work

has been done on representing and manipulating incomplete information in object-oriented models and databases. A related area is the work on versioning in object-oriented database systems [CJ92, MJC93].

8 Related work

Materialization has been intuitively perceived, with different names, by several authors but no precise semantics was defined. Coad [Coa92] presents several patterns frequently occurring in the real world; the pattern closest to materialization is called "item-description pattern". Rumbaugh et al. [RBP+91] describe two constructs similar to materialization, which they refer to as *metadata* and *homomorphisms*.

Goldstein and Storey [GS94] introduced the term materialization and characterized it informally. Most of their informal discussion automatically follows from our precise definition. They claim that their materialization cannot be defined by a combination of existing abstractions. We have shown in this paper that the semantics of our materialization can be captured with the class/metaclass correspondence, and the classification and generalization abstractions.

9 Summary and further work

This paper has presented materialization as a useful abstraction pattern for conceptual schemas to relate a class of concepts to another class of more concrete concepts. Specifically, we have given a quasi-formal definition of materialization as a combination of is-a generalization, is-of classification, and class/metaclass correspondence; we have characterized several mechanisms of attribute inheritance through materialization; we have exhibited examples that demonstrate that materialization is frequently encountered in practice; we have shown the combination of several materializations and extensions of materialization to relationships, aggregation, and constraints.

Materialization could also be defined as it is implemented, namely as an ordinary binary relationship, suitably supplemented with integrity constraints that force the desired behavior. But it is precisely this embedding of constraints into a more powerful modeling operation that represents progress in conceptual modeling, by enabling the construction of more adequate conceptual models and easing the work of the human modeler. In other words, high-level abstraction patterns like materialization replace explicit constraints, that would be needed to enforce the same behavior if the abstraction pattern was not available. Instead, the constraints have become structural (or inherent, implicit), they are built in the pattern and automatically brought to bear when the modeling construct is invoked during conceptual modeling.

We are implementing materialization in an expert system CASE tool for investigating current object-oriented analysis and design methodologies (e.g., [RBP+91, Boo94, CAB+94]). We have implemented the management of materialization at the conceptual level. We have identified several kinds of constraints, e.g., for controlling inheritance, for ensuring the semantic correctness of several related materializations or a materialization combined with other abstractions (in the spirit of [FM94] for is-a hierarchies). We have studied interesting high-level optimizations in the implementation of several materializations of the same class into different more concrete classes. We also defined operational procedures for high-level updating (for example, when the more abstract class of a materialization is deleted, its attributes can selectively migrate to the concrete class).

Another area of research is to study how the semantics of materialization can be implemented into less expressive contexts, like object oriented programming languages coupled with classical relational databases.

The present work suggests many continuations. We are studying formalizations of materialization with knowledge representation languages, particularly Telos and O-Telos[MBJK90, JJ91, JEG+94]. We want to explore more thoroughly the inheritance mechanisms associated with materialization and extensions that lead to dynamic schemas. We also wish to study the relationships between materialization and versioning. We considered relaxing the (1,1) cardinalilty on the side of the more concrete class of materialization. For example, we would like to describe the materialization of several movies on the same videotape, or, similarly, of several journal issues bound in the same physical volume. This opens the door to multiple inheritance and probably requires a more powerful aggregation abstraction than the one commonly studied.

Materialization breaks the traditional database separation of concerns between the generic and the individual. This paper has shown that a controled introduction of classification at the application level substantially enhances expressiveness in the conceptual schema. There is an endless collection of materializations to be found in most application domains.

Materialization systematically invokes the classification abstraction for application domain modeling, whereas such a meta-dimension is more often invoked as a step away from domain modeling in the literature on database conceptual modeling. This is less true for knowledge representation, where the traditional database distinction between schema and instances does not hold. Materialization thus follows the current trend in bringing knowledge representa-

tion and database conceptual modeling closer to one another.

For future database management systems, opening the door (albeit cautiously) to classification in the conceptual schema suggests a revision of the role of the central multifunction schema of traditional databases, as both the support of a conceptual model for users, and a guide and control for the operation of the database system. The latter function may well lose some its importance in the future and progressively give way to a layered and distributed dynamic schema.

Acknowledgements. Our disagreements with the treatment of materialization in [GS94] provided a vigorous initial inspiration. Useful suggestions were provided by A. Borgida, S. Brinkkemper, M. Jeusfeld, F. Manola, P. Massonet, J. Mylopoulos, and M. Sakkinen.

References

[ABD+89] M. Atkinson, F. Bancilhon, D. Dewitt, K. Dittrich, D. Maier, and S. Zdonik. The object-oriented database system manifesto. In *Proc. 1st Int. Conf. on Deductive and Object-Oriented Databases, Kyoto*, 1989.

[BCN92] C. Batini, S. Ceri, and S. Navathe. *Conceptual Database Design: An Entity-Relationship Approach.* Benjamin/Cummings, 1992.

[Bee93] Catriel Beeri. Some thoughts on the future evolution of object-oriented database concepts. In *Proceedings BTW'93 – Datenbanksysteme für Büro, Technik und Wissenschaft*, "Informatik aktuell". Springer-Verlag, 1993.

[BH88] A. Bruffaerts and E. Henin. Proof trees for negation as failure: yet another Prolog meta-interpreter. In *Logic Programming, Proceedings of the 5th International Conference and Symposium*, pages 343–358, Seattle, 1988.

[Boo94] G. Booch. *Object-Oriented Analysis and Design with Applications.* Benjamin/Cummings, second edition, 1994.

[BR84] M.L. Brodie and D. Ridjanovic. On the design and specification of database transactions. In M.L. Brodie, J. Mylopoulos, and J.W. Schmidt, editors, *On Conceptual Modelling.* Springer-Verlag, 1984.

[Bra83] Ronald Brachman. What Is-a IS and what Is-a Isn't: An analysis of taxonomic links in semantic networks. *IEEE Computer*, pages 30–36, October 1983.

[BS83] D. Bobrow and M.J. Stefik. *The LOOPS Manual.* Xerox Corp., 1983.

[CAB+94] D. Coleman, P. Arnold, S. Bodoff, C. Dollin, H. Gilchrist, F. Hayes, and P. Jeremaes. *Object-Oriented Development: The Fusion Method.* Prentice-Hall, 1994.

[CJ92] W. Cellary and G. Jomier. Consistency of versions in object-oriented databases. In F. Bancilhon, C. Delobel, and P. Kanellakis, editors, *Building an Object-Oriented Database System: the Story of O_2*, chapter 19, pages 447–462. Morgan Kaufmann, 1992.

[Coa92] P. Coad. Object-oriented patterns. *Comm. of the Assoc. for Computing Machinery*, 35(9):152–159, September 1992.

[Coi87] P. Cointe. Metaclasses are first class: the ObjVlisp model. In *Proceedings of the 2nd ACM Conference on Object-Oriented Programming Systems, Languages and Applications*, pages 156–167. ACM Press, 1987.

[Dit93] K. Dittrich. Object-oriented data model concepts. In Doğaç et al. [DOBS93].

[DOBS93] A. Doğaç, M. T. Özsu, A. Biliris, and T. Sellis, editors. *Object-Oriented Database Management Systems*, NATO ASI Series, Turkey, 1993. Springer-Verlag.

[DOO91] *Proceedings 2nd International Conference on Deductive and Object-Oriented Databases, DOOD'91*, LNCS 566, Munchen, Germany, December 1991. Springer-Verlag.

[FK85] R. Fikes and J. Kehler. The role of frame-based representation in reasoning. *Comm. of the Assoc. for Computing Machinery*, 28(9), September 1985.

[FM94] A. Formica and M. Missikoff. Correctness of ISA hierarchies in object-oriented database systems. In *Proceedings EDBT'94, International Conference on Extending Database Technology*, pages 231–244, Cambridge, UK, 1994. Springer-Verlag.

[GH93] J. Göers and A. Heuer. Definition and application of metaclasses in an object-oriented database model. In *Proc. of the 9th IEEE Int. Conf. on Data Engineering*, 1993.

[GR83] A. Goldberg and D. Robson. *Smalltalk-80: The Language and its Implementation.* Addison-Wesley, 1983.

[GS94] Robert C. Goldstein and Veda C. Storey. Materialization. *IEEE Trans. on Knowledge and Data Engineering*, 6, October 1994.

[INV91a] T. Imieliński, S. Naqvi, and K. Vadaparty. Incomplete objects — a data model for design and planning applications. In *Proc. ACM-SIGMOD Int. Conf. on Management of Data*, pages 288–297, 1991.

[INV91b] T. Imieliński, S. Naqvi, and K. Vadaparty. Querying design and planning applications. In *Proceedings 2nd International Conference on Deductive and Object-Oriented Databases, DOOD'91* [DOO91].

[JEG+94] M. Jarke, S. Eherer, R. Gallersdörfer, M.A. Jeusfeld, and M. Staudt. ConceptBase - a deductive object base manager. *Journal on Intelligent Information Systems*, 1994.

[JJ91] M. Jeusfeld and M. Jarke. From relational to object-oriented integrity specification. In *Proceedings 2nd International Conference on Deductive and Object-Oriented Databases, DOOD'91* [DOO91].

[KAN93] W. Klas, K. Aberer, and E. Neuhold. Object-oriented modeling for hypermedia systems using the VODAK modeling language. In Doğaç et al. [DOBS93].

[Kee89] S.E. Keene. *Object-Oriented Programming in COMMON LISP: A programmer's Guide to CLOS*. Addison-Wesley, 1989.

[Kim90] W. Kim. *Introduction to Object-Oriented Databases*. MIT Press, 1990.

[LAC+93] M. Loomis, T. Attwood, R. Cattell, J. Duhl, G. Ferran, and D. Wade. The ODMG object model. *Journal of Object-Oriented Programming*, June 1993.

[MBJK90] J. Mylopoulos, A. Borgida, M. Jarke, and M. Koubarakis. Telos: representing knowledge about informations systems. *ACM Trans. on Office Information Systems*, 8(4):325–362, 1990.

[MBW80] J. Mylopoulos, P. Bernstein, and H. Wong. A language facility for designing interactive, database-intensive applications. *ACM Trans. on Database Systems*, 5(2), 1980.

[MJC93] C.B. Medeiros, G. Jomier, and W. Cellary. Maintaining integrity constraints across versions in a database. In *Proceedings of the 8th Brazilian Conference on Databases*, pages 83–97, May 1993.

[MN88] P. Maes and D. Nardi, editors. *Meta-level architectures and reflection*. North-Holland, 1988.

[MP93] R. Motschnig-Pitrik. The semantics of parts versus aggregates in data/knowledge modelling. In Colette Rolland, Francois Bodart, and Corine Cauvet, editors, *Advanced Information Systems Engineering, Proceedings of the 5th International Conference, CAiSE'93*, LNCS 685, pages 352–373, Paris, France, June 1993. Springer-Verlag.

[MPM92] R. Motschnig-Pitrik and J. Mylopoulos. Classes and instances. *International Journal of Intelligent and Cooperative Information Systems*, 1(1):61–92, 1992.

[PD91] N. Paton and O. Diaz. Metaclasses in object oriented databases. In R. Meersman, W. Kent, and S. Khosla, editors, *Object oriented databases: analysis, design and construction (DS-4)*, pages 331–347. North-Holland, 1991.

[PZ94] A. Pirotte and E. Zimányi. Materialization. Technical Report RR 94-04, INFODOC, Université de Bruxelles, Belgium, 1994. Forthcoming.

[RBP+91] J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, and W. Lorensen. *Object-Oriented Modeling and Design*. Prentice-Hall, 1991.

[Rus89] D.M. Russinof. Proteus: A frame-based nonmonotonic inference system. In W. Kim and F.H. Lochovsky, editors, *Object-Oriented Concepts, Databases and Applications*, pages 127–150. ACM Press, 1989.

[Van90] P. Vandamme. *Représentation des Connaissances par Réseaux Sémantiques*. PhD thesis, Unité d'informatique, Université de Louvain, Belgium, 1990.

[ZM90] S. Zdonik and D. Maier. Fundamentals of object-oriented databases. In Stanley Zdonik and David Maier, editors, *Readings in Object-Oriented Database Systems*. Morgan Kaufmann, 1990.

[ZP92] E. Zimányi and A. Pirotte. Imperfect knowledge in databases. Technical Report RR 92-36, Unité d'informatique, Université de Louvain, Belgium, October 1992. To appear in "Uncertainty Management in Information Systems: from Needs to Solutions", A. Motro and P. Smets eds, 1994.