Formalizing Materialization Using a Metaclass Approach *

Mohamed Dahchour†

Abstract

Materialization is a powerful and ubiquitous abstraction pattern for conceptual modeling. Intuitively, it relates a class of categories (e.g., models of cars) and a class of more concrete objects (e.g., individual cars). This paper formalizes the semantics of materialization using the metaclass approach of the TELOS data model. Formulas can be uniformly attached to classes, metaclasses, and meta-attributes to enforce integrity constraints and deductive rules relevant to materialization semantics. The paper also proposes some suggestions for extending TELOS to capture some materialization semantics which cannot be represented with the available constructs.

Keywords: Object Orientation, Materialization Relationship, Metaclass, TELOS.

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. Generic relationships are powerful abstraction constructs that help narrow the gap between concepts in the real world and their representation in conceptual models. For full benefit, these relationships should be made available in object-oriented languages and systems as primitives for developing conceptual models of applications. However, before their implementation, we believe that generic relationships should be first well formalized. This formalization will eliminate

*This work is part of the YEROOS (Yet another Evaluation and Research on Object-Oriented Strategies) project, principally based at the University of Louvain. See http://yeroos.qant.ucl.ac.be.

†University of Louvain, INGI (Department of Computer Science and Engineering), 2 Place Sainte-Barbe, 1348 Louvain-la-Neuve, Belgium, e-mail: dahchour@student.fsa.ucl.ac.be

the possible ambiguities between similar relationships and will play an intermediate role between the informal description of a relationship and its factual implementation.

This paper presents a formalization of *materialization* [PZMY94]. Materialization is a powerful and ubiquitous abstraction pattern. It is a semantic relationship between a class of abstract categories (e.g., models of cars) and a class of more concrete objects (e.g., individual cars). The semantics of materialization concern both *classes* and *instances* of these classes. Consequently, the formal specification of materialization must include both the specification of the *class* and the *instance* levels in a coordinated manner [KS95]. Furthermore, constraints associated with generic relationships must be defined at the conceptual level, since they govern all instances of these relationships. We remove, therefore, the burden from the designers who otherwise would have to define these constraints for each realization of materialization.

We use the metaclass approach of TELOS, a language for representing knowledge about information systems [MBJK90], to formalize materialization. TELOS has already been used to partially formalize semantics of *partOf* [MP93] and *memberOf* [MPK96] relationships.

The metaclass approach has been used successfully to *implement* some generic relationships (see e.g., [HGPK94, KS95, GSR96]). Particularly, in our previous work [DPZ97], we have presented three metaclass approaches to implement generic relationships and in [DPZ96], we have used one of these approaches to *implement* materialization in an abstract target system. In this paper, we use the metaclass approach of TELOS for the *formalization* purpose.

The paper is organized as follows. Section 2 gives an overview of materialization. Section 3 presents the main features of the TELOS data model, relevant to our formalization. Sections 4 and 5 formalize in detail the semantics of materialization at both the class and instance levels. Section 6 summarizes and concludes the paper.

2 Materialization

This section gives an overview of the materialization relationship and of its specific attribute propagation mechanisms. More detail can be found in [PZMY94].

2.1 Intuitive definition

Intuitively, materialization relates a class of categories to a class of more concrete objects. Figure 1(a) shows a materialization relating two classes: class CarModel has two monovalued attributes (name and sticker_price) and four multivalued attributes (#doors, eng_size, auto_sound, and special_equip); class Car defines three monovalued attributes (manuf_date, serial#, and owner). CarModel represents information typically displayed in the catalog of car dealers (namely,

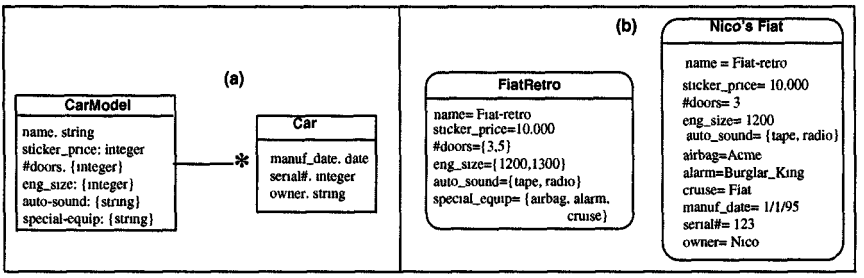


Figure 1: An example of materialization.

name and price of a car model, and lists of options for number of doors, engine size, sound equipment, and special equipment). Car represents information about individual cars (namely, manufacturing date, serial number, and owner identification). As in [PZMY94], we draw a materialization link as a straight line with a star $*$ on the side of the more concrete class.

Figure 1(b) shows an instance FiatRetro of CarModel and an instance Nico's Fiat of Car, of model FiatRetro. CarModel is the more abstract[1] class and Car is the more concrete class of materialization CarModel—*Car. Intuitively, this means 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 thus directly inherits the name and sticker_price of its model FiatRetro; this mechanism is called Type 1 attribute propagation. Nico's Fiat has attributes #doors, eng_size, and auto_sound whose values are selections among the options offered by multivalued attributes with the same name in FiatRetro; this is called Type 2 attribute propagation. For example, the value {1200,1300} of eng_size for FiatRetro indicates that each FiatRetro car comes with either eng_size = 1200 or eng_size = 1300 (e.g., 1200 for Nico's Fiat). The value {airbag, alarm, cruise_ctrl} of attribute special_equip for FiatRetro means that each car of model FiatRetro comes with three pieces of special equipment: an airbag, an alarm system, and a cruise control system. Thus, Nico's Fiat has three new attributes named airbag, alarm, and cruise_ctrl, whose suppliers are, respectively, Acme, Burglar_King, and Fiat. Other FiatRetro cars might have different suppliers for their special equipment. This mechanism is called Type 3 attribute propagation. In addition to those attributes propagated from the instance FiatRetro of class CarModel, Nico's Fiat of course has a

[1]The notion of abstractness/concreteness of materialization is distinct from the notion of abstract class of object models, where an abstract class is a class without instances, whose complete definition is typically deferred to subclasses.

value for attributes manuf_date, serial#, and owner of class Car. The semantics of attribute propagation is defined more precisely in Section 2.3.

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 materializes independently into classes VideoTape and VideoDisc (i.e., VideoTape*—Movie—*VideoDisc). VideoTapes and VideoDiscs could have attributes like inventory#, system (e.g., PAL or NTSC for VideoTape), language, availability (i.e., in-store or rented), and so on.

Materializations can also be composed in hierarchies, where the concrete class of one materialization is also the abstract class of another materialization, and so on (e.g., Play—*Setting—*Performance). For the sake of space, this paper considers only simple materialization hierarchies A—*C and abstract classes materializing in more than one concrete class as in C1*—A—*C2. A complete formalization of materialization, including composition of materializations, can be found in [Dah97].

2.2 Semi-formal semantics

We now summarize the necessary elements for a semi-formal definition of materialization. Materialization is a binary relationship (A—*C) between two classes A and C, where A is more abstract than C (or C is more concrete than A).

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

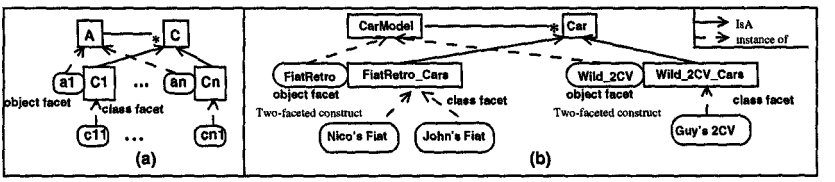


Figure 2: Semantics of materialization.

The semantics of materialization is conveniently defined as a combination of usual *is-a* (generalization) and *is-of* (classification), and of a class/metaclass correspondence. Figure 2(a) shows how the semantics of materialization A—*C is expressed with a collection of *two-faceted constructs*. Each two-faceted construct is a composite structure comprising an object, called the *object facet*, and an associated class, called the *class facet*. The object facet is an instance of the more abstract class A, while the class facet is a subclass of the more concrete

class C. The semantics of materialization induce 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 *is-a* link. They also "inherit" attributes from A, through the mechanisms of attribute propagation described in the next section. 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.

As in Figure 1, we draw classes as rectangular boxes and instances as rectangular boxes with rounded corners. Classification links (*is-of*) appear as dashed arrows, and generalization links (*is-a*) as solid arrows. To underline their double role, we draw a two-faceted construct as an object box adjacent to a class box.

Figure 2(b) sketches the basic semantics of the materialization of Figure 1(a). The FiatRetro instance of CarModel is the object facet of a two-faceted construct, whose class facet is the subclass FiatRetro_Cars of Car, describing all instances of Car with model FiatRetro. For users, Nico's Fiat and John's Fiat are instances of Car. Our semantics and its formalization describe them as ordinary instances of FiatRetro_Cars. Wild_2CV is another instance of CarModel and Guy's 2CV is an instance of class facet Wild_2CV_Cars.

2.3 Attribute propagation

Attribute propagation from the more abstract class to the more concrete class of a 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 Figure 3. The three mechanisms of attribute propagation are defined precisely as follows:

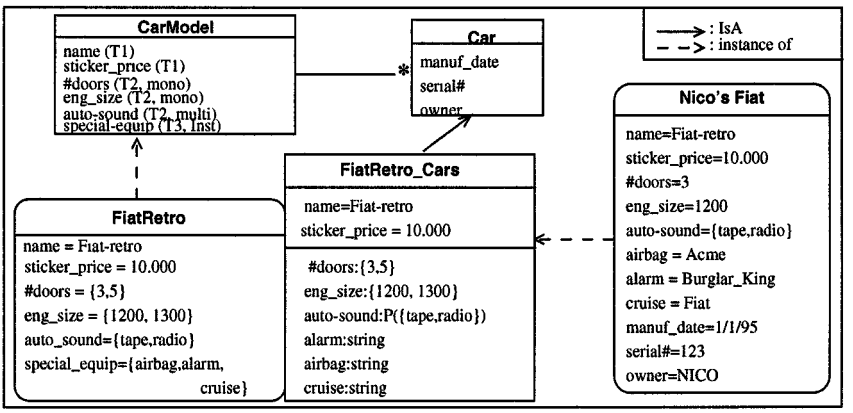


Figure 3: Attribute propagation between CarModel and Car.

1. For users, Type 1 propagation characterizes the plain transfer of an attribute value from an instance of the abstract class to 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., an attribute whose value is the same for all instances of the class facet). For example, monovalued attributes name and sticker_price of CarModel are Type 1 in materialization CarModel—*Car (see Figure 3). Their value in object facet FiatRetro (Fiat-retro and 10.000, respectively) propagates as value of class attributes with the same name in class facet FiatRetro_Cars.

2. For users, Type 2 propagation concerns multivalued attributes of the more abstract class A. Their value for an instance of A determines the type, or domain, of instance attributes with the same name, monovalued or multivalued, in the associated class facet. Again, our semantics go through abstract objects and associated class facets.

   An example of the monovalued case is exhibited by attribute eng_size of CarModel. Its value {1200,1300} for the FiatRetro object facet is the domain of values for a monovalued instance attribute with the same name eng_size of the associated class facet FiatRetro_Cars. Thus, each FiatRetro car comes either with eng_size = 1200 or with eng_size = 1300.

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

3. Type 3 propagation is more elaborate. It also concerns multivalued attributes of the more abstract class A, whose value is always a set of strings. Each element in the value of an attribute for object facet a generates a new instance attribute in the class facet associated with a. The type of generated attributes must be specified in the definition of the materialization.

   For example, attribute special_equip of CarModel propagates with Type 3 to Car. Its value {airbag, alarm, cruise_ctrl} for object FiatRetro generates three new monovalued instance attributes of type string, named airbag, alarm, and cruise_ctrl, for the associated class facet FiatRetro_Cars.

3 The TELOS data model

This section gives a general view of the main features of the TELOS data model relevant to our formalization. More details about TELOS can be found in [MBJK90]. TELOS is actually supported by the ConceptBase system [JJS96].

TELOS is a language for representing knowledge about information systems. TELOS knowledge bases are collections of *propositions*. Each proposition p is a three-tuple <from, label, to> where from, label, and to denote the source, label, and destination of the proposition, respectively. These elements can be accessed through the functions From(p), Label(p), and To(p). TELOS propositions are either *individuals* or *attributes*. *Individuals* represent what are called *objects* (e.g., the individual book OOSC2ed) and *classes* (e.g., Book) in usual object models. While *attributes* represent binary relationships between *individuals* or other relationships. An example of an attribute is [OOSC2ed, author, "B. Meyer"].

Propositions can be classified in an arbitrary number of classification levels where each proposition is an instance of one or more generic propositions called *classes*. Classes that are themselves propositions must be in their turn instances of more generic classes, and so on. For example, OOSC2ed and [OOSC2ed, author, 'B. Meyer'] are instances of Book and [Book, author, Person], respectively.

The so-called *ω-classes* can have instances along more than one level of classification. For example, Proposition has *all* propositions as instances and Class has *all* generic propositions as instances.

The following example shows the use of the different features above. The TELL operation is used to add a new proposition in the knowledge base (i.e., *create new* objects in the terminology of usual object models) or to add *new* attributes to an already defined one.

```
TELL TOKEN MT-93-#1 In BorrowedDocument WITH
   author
      firstAuthor: "C. Marcos",
      secondAuthor: "M. Cilia";
   title
      . "A SDM approach for the Prototyping of IS"
   borrowed
      : Yes;
   borrower
         "John"
   outDate
      : "05/06/97.9H"
   inDate
      : "05/06/97:18H"
END
```

```
TELL CLASS Document IN Class WITH
   attribute
      author  Person,
      title: String,
      .
END

TELL CLASS BorrowedDocument IsA Document,
      IN Class WITH
   attribute
      borrowed: String,
      borrower' Person;
      outDate  Date;
      inDate. Date;
END
```

Figure 4: TELOS definition of instances, classes, and attributes.

Figure 4 shows, on the left side, the individual document MT-93-#1 that is declared as an instance (via the IN clause) of the class BorrowedDocument defined on the right side of the figure as an instance of the metaclass Class and as a specialization of Document. The WITH clause introduces the list of attributes. For example, the two first attributes of MT-93-#1, firstAuthor and secondAuthor, are two instances of the attribute class author. The attribute [MT-93-#1, firstAuthor, "C. Marcos"] is an instance of [Document, author, Person] in exactly the same sense that MT-93-#1 is an instance of Document. The third attribute of MT-93-#1 has no external label and it is an instance of the title class. Labels of such attributes are automatically generated by the system.

In Telos, a proposition may be an instance of more than one class (multiple classification). For instance, MT-93-#1 can be an instance of both classes MasterThesisDocument and RestrictedDocument which stands for a collection of documents that are not allowed to go out the library.

Meta-attributes. The first-class status of attributes and the ability to define attributes and meta-attributes are very important in TELOS. Figure 5 shows an example of meta-attributes which are needed to define common properties of the various resource classes. These meta-attributes are introduced through the metaclass ResourceClass. In this example, source, what, available, who, from, and until are meta-attributes which may be instantiated for ResourceClass instances. The class BorrowedDocument is declared now as an instance of ResourceClass on the right side of Figure 5 and its attribute borrower is an instance of the meta-attribute who.

```
TELL CLASS ResourceClass WITH
   attribute
      source  Class,
      what·  Class,
      available.  Class,
      who  Class;
      from.  Class,
      until  Class;
END
```

```
TELL CLASS BorrowedDocument IN ResourceClass WITH
   source
      author: Person;
   what
      title. String;
   available
      borrowed. String,
   who
      borrower: Person;
   from
      outDate: Date;
   until
      inDate: Date;
END
```

Figure 5: Definition of meta-attributes.

As another example of use of meta-attributes, Figure 6 gives the definition of the meta-attribute single that restricts its instances to (at most) a single value [MBJK90]. The right side of Figure 6 shows an example of use of the meta-attribute single: we restrict the borrower of a BorrowedDocument to a single value by declaring it as an instance of single. The meta-attribute single is defined in the metaclass Class and it is inherited by BorrowedDocument by declaring BorrowedDocument as instance of Class.

Note that by default a TELOS attribute such as author: Person of Figure 5 can have *several* instances. If we want to restrict the attribute value, we have to use something like the meta-attribute single. Therefore, the declaration of attributes in TELOS should not be confused with that of the usual object data models.

```
TELL CLASS Class WITH
   attribute
      single· Class
   integrityConstraint
      single_Cnstr  $ (∀ u/Class!single)(∀ p,q/Proposition)
      (p in u) ∧ (q in u) ∧ From(p) = From(q) ⇒ (p = q) $
END
```

```
TELL CLASS BorrowedDocument IN
      ResourceClass, Class WITH
   ..
   who, single
      borrower  Person;
   ..
END
```

Figure 6: Definition of the single meta-attribute and its use.

Constraints, rules, and metaformulas. TELOS supports an assertion sub-

language to specify *integrity constraints* and *deductive rules.* Constraints and rules are formulas that appear as attribute values of propositions. They specify the behavioral part of the objects to which they are attached. Constraints are assertions that control the information supplied by users, while deductive rules are assertions that enforce new facts. For example, the integrity constraint c1 of the definition of Figure 7 ensures that the out of date for a borrowed document x must always be less than its return date. The constraint c2 ensures that a given document x cannot be borrowed by two persons at overlapping dates[2]. The deductive rule states that once a person p borrows a certain document x, the system automatically derives the fact (x.borrowed = Yes), indicating that the document is actually borrowed. The "x/C" notation is read "x is an instance of C".

```
TELL CLASS BorrowedDocument IN Class WITH
   integrityConstraint
      c1  $ (∀ x/BorrowedDocument) (x outDate ≤ x inDate)$;
      c2. $ (∀ x/BorrowedDocument)(∀ p1, p2/Person)
            (x.borrower=p1) [at t1] ∧ (x.borrower=p2) [at t2] ∧ (t1 overlaps t2) ⇒ (p1 = p2) $
   deductiveRule
      · $ (∀ x/BorrowedDocument)(∀ p/Person) (x borrower = p) ⇒ (x borrowed = Yes) $
END
```

Figure 7: Definition of constraints and deductive rules.

In traditional modeling languages, a formula is defined for a given class to constrain the behavior of only the instances of this class. In TELOS, the so-called *metaformulas* can be associated to a given metaclass to specify the behavior of both the instances of this metaclass and the instances of its instances. As an example, the constraint attached to the metaclass Class on the left side of Figure 6 is a metaformula that manipulates p and q that are instances of instances of Class!single.

To manipulate attributes and their values in definitions of formulas, we need the following functions where the time constraints are omitted [MBJK90]:

1. The dot function x.l evaluates to the set of values of the attributes of proposition x which belong to the attribute class labeled l.

2. The hat function x^l evaluates to the set of values of the attributes of proposition x with label l.

3. The bar function x|l evaluates to the set of attribute propositions with source x which are instances of the attribute class labeled l.

4. The exclamation function x!l evaluates to the set of attribute propositions with source x and label l.

[2]TELOS also supports an explicit representation of time which is not presented in this paper (see [MBJK90]).

4 Formalizing the class level semantics of materialization

In this section we formalize the class level semantics of the materialization relationship by means of two metaclasses AbstractClass and ConcreteClass that represent, respectively, abstract and concrete classes in materialization hierarchies.

```
TELL CLASS AbstractClass
     In Class WITH
  attribute
    materializes: ConcreteClass
END

TELL CLASS ConcreteClass
     In Class WITH
  attribute, single
    materOf  AbstractClass
END

TELL CLASS AbstractClass WITH
  deductiveRule
    materDedRule:
        $ (∀ A/AbstractClass)(∀ C/ConcreteClass)
        (C ∈ A.materializes) ⇒ (C materOf = A) $
END
```

Figure 8: Definition of AbstractClass and ConcreteClass metaclasses.

Figure 8 shows the definitions of the AbstractClass and ConcreteClass metaclasses. We declare AbstractClass as instance of the predefined metaclass Class. AbstractClass contains one meta-attribute whose label is materializes and destination is ConcreteClass. In the middle of Figure 8, we declare the metaclass ConcreteClass that plays the inverse role of AbstractClass. ConcreteClass contains one meta-attribute whose label is materOf and destination is AbstractClass. The materOf meta-attribute is constrained to be of a single value, meaning that a given concrete class has only one associated abstract class.

On the right side of Figure 8, we add the deductive rule materDedRule to the AbstractClass metaclass to specify that once a given class A is declared as an abstract class which materializes in a given concrete class C, the system automatically induces the fact (C.materOf = A) which means that C is a materialization of the abstract class A. A similar deductive rule can be associated with the ConcreteClass metaclass to play the dual role.

4.1 Definition of the materialization characteristics

Materialization characteristics are formalized as attributes of the meta-attribute materializes. To be able to attach properties to materializes, we have to declare this later as a *metaclass* as shown in Figure 9.

In Figure 9, we apply the "!" symbol to AbstractClass to access the attribute materializes itself. The figure shows the following characteristics: cardinality denotes the cardinality of an abstract class regarding a concrete class. The trivial associated constraint minmaxCard states that the minimal cardinality is always less than the maximal one. The remaining attributes labeled inhAttrT1, inhAttrT2, and inhAttrT3 specify propagation modes for attributes of the abstract class to the corresponding concrete class. Definitions of their destinations (i.e., domains) are given on the right side of the figure:

1. Attribute-1Def is the name of an attribute propagating with Type 1;

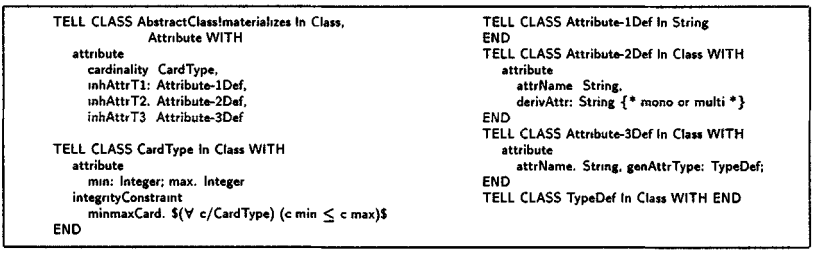


Figure 9: Definition of the materialization characteristics.

2. Attribute-2Def gives, for an attribute propagating with Type 2, its name and the kind derivedAttr (monovalued or multivalued) of the derived instance attribute [3];

3. Attribute-3Def gives the name of an attribute propagating with Type 3 and the value type genAttrType (TypeDef).

Note that the meta-attribute inhAttrT1 (resp., inhAttrT2 and inhAttrT3) can be instantiated in application with as many Type 1 (resp., Type 2 and Type 3) attributes as needed.

4.2 Example of use of generic semantics

As an example, Figure 10 shows how the CarModel—*Car materialization is formalized by invoking the generic semantics.

In Figure 10 (a), the classes CarModel and Car are created as ordinary classes, independently of the notion of materialization. To take into account the materialization relationship between CarModel and Car, we declare in Figure 10 (b) CarModel as an instance of AbstractClass and Car as an instance of ConcreteClass. During the creation of CarModel we instantiate the meta-attribute materializes of AbstractClass by materializesCar. Note that there is no need to instantiate the meta-attribute materOf of ConcreteClass during the creation of Car. This will be automatically achieved by the deductive rule materDedRule of Figure 8 (b). In the attribute CarModel!materializesCar we specify that: the cardinality of CarModel regarding Car is [0:N]; name and sticker_price propagate with Type 1; #doors and eng_size both propagate with Type 2 and each produces a monovalued instance attribute, while auto_sound produces, also with Type 2, a multivalued instance attribute; special_equip generates, with Type 3, new instance attributes of type String.

Generic semantics of materialization given above also hold for abstract classes that materialize in *more* than one concrete class such as a hierarchy of materializations C1*—A—*C2. In such a case, we create A as an instance of Ab-

[3]The {* ...*} notation denotes a TELOS comment.

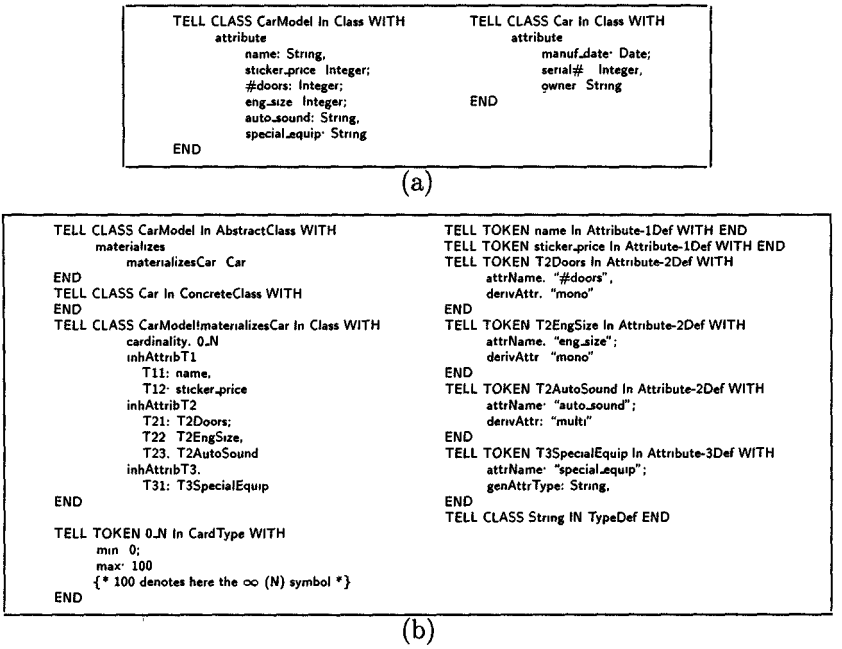


Figure 10: Materialization CarModel—∗Car.

stractClass and instantiate the attribute materializes into two attributes materializesC1 and materializesC2 with destinations C1 and C2, respectively. Then, we declare A!materializesC1 and A!materializesC2 as instances of the metaclass Class to capture different characteristics of materializations A—∗C1 and A—∗C2, respectively.

4.3 Constraints related to inheritance attributes

This section defines two constraints related to inheritance attributes. The first one expresses the membership of inheritance attributes to abstract classes and the second states that Type 2 attributes must be multivalued.

Inheritance attributes are members of abstract classes. All inheritance attributes appearing in the definition of the meta-attribute AbstractClass!materializes must be attributes of the more abstract class. For instance, the Type 1 attributes (name, sticker_price), the Type 2 attributes (#doors, eng_size, and auto_sound), and the Type 3 attribute (special_equip) of the Figure 10 (b) must be attributes of the abstract class CarModel. Figure 11 shows, respectively, the constraints T1Attr_Cnstr, T2Attr_Cnstr, and T3Attr_Cnstr that express the membership of the inheritance attributes to the abstract class of a materialization relationship.

```
TELL CLASS AbstractClass!materializes
  WITH integrityConstraint
   T1Attr_Cnstr
     $ (∀ M/AbstractClass!materializes)
     (∀ A/AbstractClass) From(M)=A ⇒
     (∀ T1/Attribute-1Def)(∀ S1/String)
     (T1 ∈ M.inhAttrT1 ∧ To(T1)=S1
     ⇒ (∃ p/Proposition)
     (p ∈ A | attribute) ∧ Label(p)=S1 $
  END

TELL CLASS AbstractClass!materializes
  WITH integrityConstraint
   T2Attr_Cnstr:
     $ (∀ M/AbstractClass!materializes)
     (∀ A/AbstractClass) From(M)=A ⇒
     (∀ T2/Attribute-2Def)(∀ S2/String)
     (T2 ∈ M inhAttrT2) ∧ (S2 = T2 attrName)
     ⇒ (∃ p/Proposition)
     (p ∈ A | attribute) ∧ Label(p)=S2 $
  END

TELL CLASS AbstractClass!materializes
  WITH integrityConstraint
   T3Attr_Cnstr:
     $ (∀ M/AbstractClass!materializes)
     (∀ A/AbstractClass) From(M)=A ⇒
     (∀ T3/Attribute-3Def)(∀ S3/String)
     (T3 ∈ M inhAttrT3) ∧ (S3 = T3 attrName)
     ⇒ (∃ p/Proposition) (p ∈ A | attribute)
     ∧ Label(p)=S3 $
  END
```

Figure 11: Membership metaconstraints of inheritance attributes.

Type 2 attributes are multivalued. Further the constraints related to membership of the inheritance attributes to the abstract classes, Type 2 attributes (e.g., #doors, eng_size, and auto_sound of Figure 10 (b)) must be *multivalued*: they must have *more* than one value at the instance level (e.g., in FiatRetro of Figure 3, #doors has two values 1200 and 1300).

```
TELL CLASS AbstractClass WITH
   integrityConstraint
    T2attr_are_multivalued:
      $ (∀ A/AbstractClass)(∀ M/AbstractClass!materializes)
      From(M)=A ⇒
      (∀ T2/Attribute-2Def)(∀ S2/String)
      (T2 ∈ M.inhAttrT2) ∧ (S2 = T2.attrName)
      ⇒ (∃ p/Proposition) (p ∈ A | attribute) ∧ Label(p)=S2
      ⇒ (p in Class!multivalued)
      $
  END

TELL CLASS Class WITH
   attribute
    multivalued  Class
   integrityConstraint
    multivalued_Cnstr.
      $ (∀ attm/Class!multivalued)(∃ p,q/Proposition)
      (p in attm) ∧ (q in attm) ∧ From(p) = From(q)
      ⇒ (p ≠ q)
      $
   END
```

Figure 12: Metaconstraint related to values of Type 2 inheritance attributes.

The left side of Figure 12 shows the definition of the constraint ensuring the above requirements. The Type 2 attributes are declared as instances of the meta-attribute Class!multivalued we define on the right side of Figure 12. This figure shows that the source and the destination of the attribute labeled multivalued are of type Class. The constraint associated with multivalued states that for every instance attm of Class!multivalued, there are at *least* two distinct instances p and q with common source[4].

According to the constraint of Figure 12, Figure 10(a) must be revised: the Type 2 attributes (#doors, eng_size, and auto_sound) must have been declared as instances of the meta-attribute multivalued.

4.4 Cardinality constraints

In this section we formalize the cardinality constraints of materialization.

Definition of the [0:N] and [1:1] cardinalities. Figure 13 shows definitions of two constraints card0_NCnstr and card1_1Cnstr which formalize, respectively,

[4]The meta-attribute multivalued is defined in the same spirit as the meta-attribute single (see Figure 6).

the cardinalities [0:N] associated with abstract classes and [1:1] associated with concrete classes. In Figure 13, we manipulate the abstract and concrete *objects* (e.g., a and c) that are, respectively, instances of AbstractObject and ConcreteObject classes we define in the next section. The last implication of the constraint card0_NCnstr is always evaluated to TRUE, meaning that we tolerate both the existence and the absence of a concrete instance c for a given abstract object a. The card1_1Cnstr definition is composed of two parts: the *existence* part which states that for each concrete object there is an associated abstract object and the *uniqueness* part stating that there is one and only one associated abstract object.

```
TELL CLASS AbstractClass!materializes WITH
   integtityConstraint
    card0_NCnstr.
      $ (∀ M/AbstractClass!materializes)
      (∀ A/AbstractClass)(∀ C/ConcreteClass)
      From(M)=A ∧ To(M)=C ∧ (M.cardinality = 0_N)
      ⇒ (∀ a/AbstractObject) (a in A) ⇒
      (∃ c/ConcreteObject) (c in C) ∧ (c ∈ a.materializes)
      ⇒ (TRUE)
      $
END
TELL TOKEN 0_N In CardType WITH
    min: 0, max. 100
   {* 100 denotes here the ∞ (N) symbole *}
END
```

```
TELL CLASS ConcreteClass!materOf WITH
   integtityConstraint
    card1_1Cnstr
      $ (∀ M/ConcreteClass!materOf)(∀ A/AbstractClass)(∀ C/ConcreteClass)
      From(M)=C ∧ To(M)=A ∧ (M cardinality = 1_1) ⇒
      (∀ c/ConcreteObject) (c in C) ⇒
      (∃ a/AbstractObject) (a in A) ∧ (a ∈ c materOf) {* existence *} ∧
      (∀ a1, a2/AbstractObject) (a1 in A) ∧ (a1 ∈ c materOf) ∧ (a2 in A) ∧
      (a2 ∈ c.materOf) ⇒ (a1 = a2)) {* uniqueness *} $
END
TELL TOKEN 1_1 In CardType WITH
      min: 1,
      max: 1
END
```

Figure 13: Definition of the [0:N] and [1:1] cardinality constraints.

Definition of the $[C_{min}, C_{max}]$ cardinality. As for the $[C_{min}, C_{max}]$ cardinality (e.g., [3,6]), which may be associated at the side of abstract classes, there is no way, in Telos, for its formalization. To deal with this category of cardinality, we define a new predicate Length(X, Class, Assertion, Integer) with the following meaning: the last argument is the number of elements X, instances of the second argument, satisfying the assertion given in the third argument. Formally, we can write:

$$\text{Length}(x, C, \mathcal{P}, N) \equiv \#\{x/C \text{ Holds}(\mathcal{P}(x))\} = N$$

The predicate Holds is true whenever its argument is an assertion that follows from the knowledge base [MBJK90]. The "#" symbol is applied to a given set and denotes the number of its elements.

As a result of this new predicate, the constraint related to the $[C_{min}, C_{max}]$ cardinality will be as shown in Figure 14.

```
TELL CLASS AbstractClass!materializes WITH
   integtityConstraint
    cardMin_MaxCnstr. $ (∀ M/AbstractClass!materializes) (∀ A/AbstractClass)(∀ C/ConcreteClass)
      From(M)=A ∧ To(M)=C ∧ 1≤M.cardinality.min≤M cardinality.max ∧ M.cardinality max≥2
      ⇒ (∀ a/AbstractObject) (a in A) ∧ (a in AbstractObject) ⇒
      Length(c, ConcreteObject, (c in C) ∧ (c ∈ a materializes), K) ∧ (Min_Max.min ≤ K ≤Min_Max.max) $
END
```

Figure 14: Definition of the constraint related to the cardinality $[C_{min}, C_{max}]$.

5 Formalizing the instance level semantics of materialization

This section describes the instance level semantics of materialization by means of two classes AbstractObject and ConcreteObject that represent, respectively, abstract objects and concrete objects, instances of instances of the metaclasses AbstractClass and ConcreteClass, respectively.

```
TELL CLASS AbstractObject
        In Class WITH
   attribute
      classFacet: Class
      materializes  ConcreteObject
END
```

```
TELL CLASS ConcreteObject
        In Class WITH
   attribute, single
      materOf  AbstractObject
END
```

```
TELL CLASS AbstractObject WITH
   deductiveRule
      materDedRule:
      $ (∀ a/AbstractObject)(∀ c/ConcreteObject)
            (c ∈ a.materializes) ⇒ (c materOf = a) $
END
```

Figure 15: Definitions of AbstractObject and ConcreteObject classes.

The definitions of the classes AbstractObject and ConcreteObject are depicted in Figure 15. On its left side, the figure shows that each abstract object has an associated *class facet*, denoted classFacet. An abstract object has also an attribute materializes whose domain is ConcreteObject and each concrete object has an attribute materOf whose domain is AbstractObject.

For reason of uniformity, the same names materializes and materOf are used both for the description of the *class* level and the *instance* level of materialization semantics.

The attribute materOf is declared as an instance of single to assert that a concrete object is materialization of only one abstract object. Finally, as in Figure 8, we give on the right side of Figure 15 a deductive rule that automatically derives from the fact (c ∈ a.materializes) another fact (c.materOf = a) indicating that c is a materialization of a.

5.1 Constraints related to the abstract objects

Figure 16 shows two constraints concerning the abstract objects. On the left side we define the abstractObj_Cnstr constraint which expresses that instances of instances of AbstractClass must be instances of AbstractObject. On the right side, the ObjClassFacet_Cnstr constraint means that each abstract object a has one and only one (denoted by the "∃!" symbole) associated class facet regarding a given materialization. The constraint also shows that the class facet must be a subclass of the concrete class C that is a materialization of the abstract class A, the class of a.

The ObjClassFacet_Cnstr constraint implicitly means that, each time the user creates an abstract instance a, he/she also must create *explicitly* its associated class facet to ensure the constraint. We think that it would be more reasonable and more natural to *implicitly* generate the class facet of a given abstract object. For this, we propose to extend the definition of the *deductive rule*. The actual

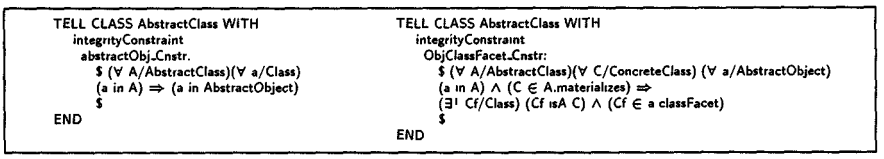
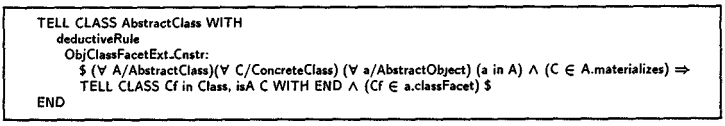


Figure 16: Constraints associated with the abstract objects.

definition of a deductive rule allows only the possibility to derive facts that are *logical expressions*. The extension we propose consists of the specification, on the right side of a deductive rule, of some *operations* we wish to be executed at run time as for the rule-based system MARVEL [KFP88]. Such an extended rule will be of the pattern: $< preconditions > \Rightarrow < usualfacts >$ [and $<$ *operations* $>$] where $< usualfacts >$ is the ordinary part we find on the right side of a usual deductive rule and $< operations >$ stands for operations we wish to be automatically executed by the system when the part $< preconditions >$ is satisfied.

The ObjClassFacet_Cnstr *constraint* on the right side of Figure 16 becomes the following extended *deductive rule*:



This deductive rule means that for each abstract object a, instance of an abstract class A which materializes in C, the system automatically generates a class Cf as an instance of Class and as a subclass of C. The system induces then the fact (Cf ∈ a.classFacet) meaning that Cf is a potential class facet of a.

5.2 Constraints related to the concrete objects

Figure 17 shows two constraints regarding the concrete objects. On the left side, the concObj_Cnstr constraint expresses that instances of instances of ConcreteClass must be instances of ConcreteObject. On the right side, the concObjClassFacet_Cnstr constraint expresses that all concrete objects c, instances of a given concrete class C, and materializing a given abstract object a, must be instances of the class facet associated with a. For example, Nico's Fiat that is instance of Car and that materializes FiatRetro must be instance of FiatRetro_Cars, the class facet associated with FiatRetro (see Figure 2).

```
TELL CLASS ConcreteClass WITH
   integrityConstraint
    concObj_Cnstr
      $ (∀ C/ConcreteClass)(∀ c/Class)
      (c in C) ⇒ (c in ConcreteObject)
      $
END
```

```
TELL CLASS ConcreteClass WITH
   integrityConstraint
    concObjClassFacet_Cnstr:
      $ (∀ A/AbstractClass)(∀ C/ConcreteClass) (C ∈ A.materializes) ⇒
      (∀ c/ConcreteObject) (∀ a/AbstractObject) (a in A) ∧
      (Cf ∈ a.classFacet) ∧ (Cf isA C) ∧ (c in C)
      ∧ (c ∈ a materializes) ⇒ (c in Cf)
      $
END
```

Figure 17: Constraints associated with the concrete objects.

5.3 Constraints related to attribute propagation

Propagation of Type 1 attributes. Figure 18 shows, on its left side, the constraint T1AttrInClassFacet_Cnstr which imposes the Type 1 attributes of an abstract class A to be attributes of each class facet associated with each instance a of A. The right side of Figure 18 shows the constraint T1AttrAreClassAttr_Cnstr which states that Type 1 attributes are *class attributes*[5] in class facets: for a given Type 1 attribute T1, if the value of T1 in a given abstract object a is v then all the instances of the class facet associated with a will come with the same value v. Another alternative to deal with Type 1 attributes is the use of delegation mechanism [Lie86] that would permit concrete instances to access directly the Type 1 attribute values in the abstract instance a, without storing it redundantly in class facets. Unfortunately, TELOS does not provide facilities for using the delegation mechanism.

```
TELL CLASS AbstractClass WITH
   integrityConstraint
    T1AttrInClassFacet_Cnstr:
      $ (∀ M/AbstractClass!materializes) (∀ A/AbstractClass)
      (∀ C/ConcreteClass)(∀ T1/Attribute-1Def) (∀ S1/String)
      (∀ D/Domain)
      (C ∈ A.M) ∧ (T1 ∈ M.inhAttrT1) ∧ To(T1)=S1
      ⇒ (∃ p/Proposition) (p ∈ A | attribute) ∧ Label(p)=S1
      ∧ To(p)=D ⇒ (∀ a/AbstractObject) (∀ Cf/Class)
      (a in A) ∧ (Cf ∈ a.classFacet) ∧ (Cf isA C) ⇒
      (∃ q/Proposition) (q ∈ Cf | attribute) ∧
      Label(q)=S1 ∧ To(q)=D
      $
END
```

```
TELL CLASS AbstractClass WITH
   integrityConstraint
    T1AttrAreClassAttr_Cnstr:
      $ (∀ M/AbstractClass!materializes) (∀ A/AbstractClass)
      (∀ C/ConcreteClass)(∀ T1/Attribute-1Def) (∀ S1/String)
      (∀ D/Domain) (C ∈ A.M) ∧ (T1 ∈ M.inhAttrT1) ∧ To(T1)=S1
      ⇒ (∃ p/Proposition) (p ∈ A | attribute) ∧ Label(p)=S1
      ∧ To(p)=D ⇒ (∀ a/AbstractObject) (∀ Cf/Class)
      (a in A) ∧ (Cf ∈ a classFacet) ∧ (Cf isA C)
      ⇒ (∀ q/Proposition) (q ∈ Cf | attribute) ∧
      Label(q)=S1 ∧ To(q)=D ⇒
      (∀ u,v,c/Proposition) (c in Cf) ∧ (u in q) ∧
      From(u)=c ∧ To(u)=v ⇒ (v = a.S1) $
END
```

Figure 18: Constraints associated with Type 1 attribute propagation.

Propagation of Type 2 attributes. The constraint T2AttrInClassFacet_Cnstr of the left side of Figure 19 imposes the Type 2 attributes of an abstract class A to be also attributes of each class facet associated with each instance a of A.

The domain of the Type 2 attributes in class facets is the *same* as in the abstract class, but it will be, next, *restricted* to only a set of *fixed* values. The

[5] we mean here the usual sense of "class attributes" as opposed to "instance attributes".

```
TELL CLASS AbstractClass WITH
   integrityConstraint
     T2AttrInClassFacet_Cnstr:
       $ (∀ M/AbstractClass!materializes) (∀ A/AbstractClass)
       (∀ C/ConcreteClass)(∀ T2/Attribute-2Def)
       (∀ S2/String)(∀ D/Domain)
       (C ∈ A.M) ∧ (T2 ∈ M.inhAttrT2) ∧ (S2 = T2.attrName)
       ⇒
       (∃ p/Proposition)
       (p ∈ A | attribute) ∧ Label(p)=S2 ∧ To(p)=D
       ⇒ (∀ a/AbstractObject)(∀ Cf/Class)
       (a in A) ∧ (Cf ∈ a.classFacet) ∧ (Cf isA C)
       ⇒ (∃ q/Proposition) (q ∈ Cf | attribute) ∧
       Label(q)=S2 ∧ To(q)=D $
END
```

```
TELL CLASS AbstractClass WITH
   integrityConstraint
     T2ValuePropag_Cnstr:
       $ (∀ M/AbstractClass!materializes)(∀ A/AbstractClass)
       (∀ C/ConcreteClass) (∀ T2/Attribute-2Def)(∀ S2/String)
       (C ∈ A M) ∧ (T2 ∈ M.inhAttrT2) ∧ (S2 = T2.attrName) ⇒
       (∀ a/AbstractObject)(∀ Cf/Class)(∀ q/Proposition)
       (a in A) ∧ (Cf ∈ a classFacet) ∧ (Cf isA C) ∧
       (q ∈ Cf | attribute) ∧ Label(q)=S2 ⇒
       [ (T2 derivAttr = "mono") ⇒ (q in Class!single) ∧
       ((∀ u,v,c/Proposition) (c in Cf) ∧ (u in q) ∧ From(u)=c ∧ To(u)=v ⇒
       (v ∈ a.S2) ) ]        ∧
       [ (T2. derivAttr = "multi") ⇒
       (∀ u,v,c/Proposition) (c in Cf) ∧ (u in q) ∧ From(u)=c ∧ To(u)=v ⇒
       (v ∈ a S2) ] $
END
```

Figure 19: Constraints associated with Type 2 attribute propagation.

constraint responsible of this restriction is T2ValuePropag_Cnstr as given on the right side of Figure 19. This constraint expresses that for each concrete instance c, instance of a class facet Cf, the value of its Type 2 attribute S2 must belong to the set of values of S2 in the abstract object a that materializes in c. If S2 is monovalued, then S2 must have only one value in c, otherwise S2 can have several values.

For instance, the Type 2 attribute "#doors:Integer" of CarModel must be also an attribute of the class facet FiatRetro_Cars associated with FiatRetro, instance of CarModel. The domain of #doors in FiatRetro_Cars is also Integer, but its value in Nico's Fiat, instance of FiatRetro_Cars, must be restricted to either 3 or 5: the two possible values fixed by the abstract instance FiatRetro.

Propagation of Type 3 attributes. Figure 20 shows the T3ValuePropag_Cnstr constraint that formalizes the propagation of Type 3 attributes. It shows that each value V3 of a given Type 3 attribute in an abstract object a is a new attribute of the class facet Cf associated with a. The domain of V3 in Cf is D which a user can supply in advance by using the attribute genAttrType associated with the definition of Type 3 attribute (Attribute-3Def).

```
TELL CLASS AbstractClass WITH
   integrityConstraint
     T3ValuePropag_Cnstr
       $ (∀ M/AbstractClass!materializes)(∀ A/AbstractClass)
       (∀ C/ConcreteClass)(∀ T3/Attribute-3Def) (∀ S3/String)(∀ D/TypeDef)
       (C ∈ A M) ∧ (T3 ∈ M inhAttrT3) ∧ (S3 = T3.attrName) ∧ (T3.genAttrType = D) ⇒
       (∃ p/Proposition) (p ∈ A | attribute) ∧ Label(p)=S3 ⇒
       (∀ a/AbstractObject)(∀ Cf/Class) (∀ V3/Class) (a in A) ∧ (Cf ∈ a.classFacet) ∧ (V3 ∈ a S3)
       ⇒ (∃ q/Proposition) (q ∈ Cf | attribute) ∧ Label(q)=V3 ∧ To(q)=D
       $
END
```

Figure 20: Constraints associated with Type 3 attribute propagation.

6 Conclusion

This paper has presented a formalization of materialization, a new and ubiquitous abstraction pattern relating a class of abstract categories and a class of more concrete objects. Materialization allows the definition of new and powerful attribute propagation (or inheritance) mechanisms from the abstract class to the concrete class.

Our formalization was carried out using the metaclass approach of TELOS. Two *metaclasses* AbstractClass and ConcreteClass were built as templates to capture the semantics of materialization at the class level and two additional *classes* AbstractObject and ConcreteObject were defined to capture the semantics of materialization at the instance level.

The metaclasses AbstractClass and ConcreteClass define the meta-attributes materializes and materOf, respectively. Thanks to the *class* status of TELOS *attributes*, the meta-attributes materializes and materOf are declared as ordinary metaclasses to which we attached all characteristics of materialization relationship.

Various constraints have been uniformly defined at the levels of classes, metaclasses, and attributes to ensure the semantics of materialization at both the class and the instance levels in a coordinated manner.

The metaclass approach allowed us to define the semantics of materialization as a unit which is independent of any implementation consideration. Consequently, our formalization can be used to *implement* materialization in various systems.

Although our formalization is more suitable for implementation systems that support metaclasses, it can be also used to assist an implementation in non metaclass-based systems. Furthermore, our approach can be followed to formalize other new generic relationships.

This work had also demonstrated the power of the TELOS language to express the requirements related to the materialization semantics. To fully formalize materialization, we proposed two suggestions to extend TELOS. One consisted of the definition of a new predicate required to formalize the cardinality $[C_{min}, C_{max}]$. The second one consisted of the extension of the deductive rule definition for specifying some operations we would wish to be automatically executed by the system when given preconditions are satisfied.

Our work has several continuations. An interesting one deals with the formalization of the common semantics of a large repository of new *generic relationships* (e.g., *Member-Of* [MPS95], *Role-Of* [WDJS94], *Part-Of* [KP97], *Ownership* [HPYG95]). Another continuation will be to start from our formalization and try to realize an effective implementation in a target system (e.g., ConceptBase) in such manner that it will be possible to evaluate this system with regard to its power in the implementation of materialization semantics.

Acknowledgements. I wish to thank all the colleagues in the YEROOS

project for their critical comments on an earlier draft of this article and particularly M. Guy Fokou. I am also grateful to Christoph Quix (RWTH Technical University of Aachen) for many useful discussions about TELOS and ConceptBase. Finally, I thank Professor Matthias Jarke (RWTH Technical University of Aachen) for the permission to use the ConceptBase system.

References

[Dah97] M. Dahchour. Formalizing materialization in the TELOS data model. Technical Report TR-97/28, IAG-QANT, Université catholique de Louvain, Belgium, November 1997.

[DPZ96] M. Dahchour, A. Pirotte, and E. Zimányi. Metaclass implementation of materialization. Technical Report YEROOS TR-96/06, January 1996. Submitted for publication.

[DPZ97] M. Dahchour, A. Pirotte, and E. Zimányi. Metaclass implementations of generic relationships. Technical Report YEROOS TR-97/25, 1997. Submitted for publication.

[GSR96] G. Gottlob, M. Schrefl, and B. Röck. Extending object-oriented systems with roles. *ACM Trans. on Office Information Systems*, 14(3):268–296, 1996.

[HGPK94] M. Halper, J. Geller, Y. Perl, and W. Klas. Integrating a part relationship into an open OODB system using metaclasses. In N.R. Adam, B.K. Bhargava, and Y. Yesha, editors, *Proc. of the 3rd Int. Conf. on Information and Knowledge Management, CIKM'94*, pages 10–17, Gaithersburg, Maryland, November 1994. ACM Press.

[HPYG95] M. Halper, Y. Perl, O. Yang, and J. Geller. Modeling business applications with the OODB ownership Relationship. In *Proc. of the 3rd Int. Conf. on AI Applications on Wall St.*, pages 2–10, June 1995.

[JJS96] M. Jarke, M.A. Jeusfeld, and M. Staudt. *ConceptBase V4.1 User Manual.* 1996.

[KFP88] G. E. Kaiser, P. H. Feiler, and S. S. Popovich. Intelligent assistance for software development and maintenance. *IEEE Software*, 5(3):40–49, 1988.

[KP97] M. Kolp and A. Pirotte. An aggregation model and its C++ implementation. In *Proc. of the 4th Int. Conf. on Object-Oriented Information Systems*, Brisbane, Australia, 1997. To appear.

[KS95] W. Klas and M. Schrefl. *Metaclasses and their application.* LNCS 943. Springer-Verlag, 1995.

[Lie86] H. Lieberman. Using prototypical objects to implement shared behavior in object oriented systems. In N.K. Meyrowitz, editor, *Proc. of the Conf. on Object-Oriented Programming Systems, Languages and Applications, OOPSLA'86*, pages 214–223, Portland, Oregon, November 1986. ACM SIGPLAN Notices 21(11), 1986.

[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.

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

[MPK96] R. Motschnig-Pitrik and J. Kaasboll. Part-whole relationship categories and their application in object-oriented analysis. In *Proc. of the 5th Int. Conf. on Information System Development, ISD'96*, September 1996.

[MPS95] R. Motschnig-Pitrik and V.C. Storey. Modelling of set membership: The notion and the issues. *Data & Knowledge Engineering*, 16(2):147–185, 1995.

[PZMY94] A. Pirotte, E. Zimányi, D. Massart, and T. Yakusheva. Materialization: a powerful and ubiquitous abstraction pattern. In J. Bocca, M. Jarke, and C. Zaniolo, editors, *Proc. of the 20th Int. Conf. on Very Large Databases, VLDB'94*, pages 630–641, Santiago, Chile, 1994. Morgan Kaufmann.

[WDJS94] R. Wieringa, W. De Jonge, and P. Spruit. Roles and dynamic subclasses: a modal logic approach. In M. Tokoro and R. Pareschi, editors, *Proc. of the 8th European Conf. on Object-Oriented Programming, ECOOP'94*, LNCS 821, pages 32–59, Bologna, Italy, July 1994. Springer-Verlag.