Inheritance under Participation Constraints and Disjointness

Anastasia Analyti[1], Nicolas Spyratos[3]*, Panos Constantopoulos[1,2], Martin Doerr[1]

[1] Institute of Computer Science, Foundation for Research and Technology-Hellas, Greece
[2] Department of Computer Science, University of Crete, Greece
[3] Laboratoire de Recherche en Informatique, Universite de Paris-Sud, France
E-mail: analyti, panos, martin@ics.forth.gr, spyratos@lri.fr

Abstract

Traditional database systems have been successful in handling large amounts of data but lack mechanisms for schema derivations and schema verification. In this paper, we propose (i) structures that carry expressive and useful information on the database schema, (ii) a set of inference rules for schema derivations, and (iii) a mechanism for discovering contradictory schema declarations.

To this end, in addition to the usual *Isa* relation, we define a stronger form of specialization for properties, that we call restriction isa, or *Risa* for short. The *Risa* relation expresses property value refinement. A distinctive feature of our model is that it supports the interaction between (explicit or derived) *Isa* and *Risa* relations, in different contexts. We demonstrate that the combination of *Isa* and *Risa* provides a powerful conceptual modeling mechanism.

The *Risa* relation allows to express participation constraints on properties. Specifically, properties of a class are characterized as *necessary*, *possible*, or *inapplicable* on a given subclass. Whether explicitly declared or derived, this information is useful for several reasons: (i) It aids the user to understand better the semantics of the subclass, (ii) it uncovers contradictory declarations or design errors, and (iii) it characterizes property values that are missing from the database.

Our data model supports property inheritance, disjoint classes, and negative information (expressed in the form of inapplicable properties). In particular, disjointness, in conjuction with *Isa* and *Risa*, can lead to the derivation of negative information or the detection of contradictory declarations.

Keywords: Property inheritance, participation constraints, disjointness, negative information, schema derivations, conceptual modeling.

1 Introduction

Traditional database systems have been successful in handling large amounts of data but lack mechanisms for schema derivations and schema verification. Information systems should provide guidance and help users determine their informational needs, by navigating through

*Research conducted while this author was visiting with the Institute of Computer Science, Foundation for Research and Technology-Hellas.

the conceptual schema and by querying the structure of the data [30]. Many user queries seek knowledge from the conceptual schema and have little to do with the individual objects stored in the database. To satisfy this need, we propose (i) structures that carry useful and expressive information on the database schema, (ii) a set of inference rules for schema derivations, and (iii) a mechanism for discovering contradictory schema declarations.

In semantic and object-oriented data models, specialization is an important conceptual modeling abstraction, expressed through the Isa relation. Though this relation seems to be sufficient to express specialization between entity classes, an additional relation is required in order to express specialization between properties. In this paper, in addition to Isa, we define a stronger form of specialization that we call restriction isa, or $Risa$ for short. The $Risa$ relation is defined for properties and expresses value refinement. We demonstrate that the combination of Isa and $Risa$ provides a powerful conceptual modeling mechanism.

To get a feeling of what the $Risa$ relation is, consider the entity class *Art collector* with a property *collects* that takes values in *Art object*, and the entity class *Painting collector* with a property *collects* that takes values in *Painting*. The classes *Painting collector* and *Painting* are subclasses of *Art collector* and *Art object*, respectively. The information that *some* of the art objects collected by a painting collector are paintings can be expressed through the usual Isa relation between the two *collects* properties. However, to express that *all* art objects collected by a painting collector are paintings, a stronger form of Isa is needed that represents property refinement. It is precisely this stronger form of Isa that we call $Risa$. In our model, Isa and $Risa$ relations are either declared by the user or derived by a set of inference rules. Derived Isa and $Risa$ relations enrich the original schema with information not explicitly declared.

Regarding properties of classes, we characterize them with respect to the participation of class instances in the property. Specifically, a property a of a class can be *necessary*, *possible*, or *inapplicable* on a subclass c. The property a is called *necessary* on c, if every instance of c has the property a. For example, the property *flying* of class $Bird$ is a necessary property on the subclass *Sea gulls*. The property a is called *possible* on c, if every instance of c can have the property a, though currently some instances of c might not have the property. For example, the property *employment* of class $Person$ is possible on the subclass *Healthy adult*. The property a is called *inapplicable*, if no instance of c can have the property a. Intuitively, inapplicable properties express "negative" information. For example, the property *flying* of class *Bird* is inapplicable on the subclass *Penguin*. This expresses that penguins do not fly[1].

The information that a property of a class is necessary, possible, or inapplicable on a subclass c is expressed using the *Risa* relation. Such information is either declared by the user or derived by a set of inference rules. Whether explicit or derived, this information is useful for several reasons:

1. It carries important information regarding the semantics of c, and therefore aids the user to understand better the meaning of class c. For example, consider the class *Bar* and its property *serves-alcohol*. If this property is declared as necessary on a subclass c then we know that all instances of c are alcohol-serving bars. If *serves-alcohol* is declared as possible on c then we know that all instances of c are licensed-bars (though some of them may not serve alcohol currently). Finally, if *serves-alcohol* is declared as inapplicable on c then we know that all instances of c are non-licensed-bars.

2. It can help uncover contradictory declarations or design errors. For example, the derivation that a property is both possible and inapplicable on a subclass c, implies that contradictory schema declarations have been made and an error message should be given to the user.

3. It can help in characterizing property values that are missing[2] from the database. For example, suppose we know that a property a is necessary on c and that an instance of c is represented in the database but its a-value is missing. Then, we can infer that the missing value exists in the real world but it is currently unknown. Similarly, if a is inapplicable on c, then we can infer that the missing value does not exist in the real world, and will never exist. Finally, if a is possible on c, then we can infer that the missing value may or may not exist currently in the real world, but can exist at some point in time. As missing values are used to express incomplete information, the characterization of missing values is an important issue [41].

Inheritance is an important form of schema derivation in knowledge representation systems [20, 32], as well as, in object-oriented and in some extensible database systems [24, 11].

[1] Roughly speaking, the term *inapplicable* means meaningless.

[2] In the area of databases, missing values are commonly referred to as *null* values.

Yet, many of these systems define property inheritance in a procedural way [5, 10, 2, 17, 36]. Thus, they are weak in terms of both expressive power and deductive capability. In this paper, we formally define property inheritance based on the $Risa$ relation, and we consider properties inherited by a class as first-class properties of the class. Through inference rules, inherited properties are related to other properties with Isa and $Risa$ relations. These relations refine the value domain of the inherited properties. In addition, they can be used to derive that an inherited property is necessary, possible, or inapplicable. Such derivations produce useful information about the semantics of the class inheriting the property.

In summary, the main contributions of this paper are:

1. Support for declaring properties of a class as necessary, possible, or inapplicable on a subclass.

2. Schema derivations based on Isa and $Risa$ relations, class disjointness[3], and information provided by necessary, possible, or inapplicable properties.

3. Detection of contradictory schema declarations.

The operational context of our work is the Semantic Index System (SIS) [12, 13, 14, 15]. Intended applications include models for descriptive sciences, cultural documentation, thesauri, and other reference information and repository indexes. Information in the SIS is represented by an object-oriented semantic network, where nodes and links are treated uniformly, and multiple instantiation, multiple specialization, and virtually unlimited instantiation levels are supported.

The rest of the paper is organized as follows: Section 2 describes our view of real world objects. Section 3 presents the definitions of In and Isa relations. Section 4 motivates and defines the restriction isa relation. Section 5 deals with necessary, possible, and inapplicable properties. In Section 6, we define inheritance and show how Isa and $Risa$ relations result in the refinement of the value domain for the inherited properties. In Section 7, we discuss disjoint classes and how they contribute to the derivation of negative information, and in the detection of contradictory schema declarations. Section 8 discusses related work, and section 9 contains concluding remarks.

[3] Two classes are called *disjoint*, if they have no instance in common.

2 Objects

In this section, we present our view of real world objects. Then, we describe these objects in terms of *intension* and *extension*. For examples refer to Figure 1.

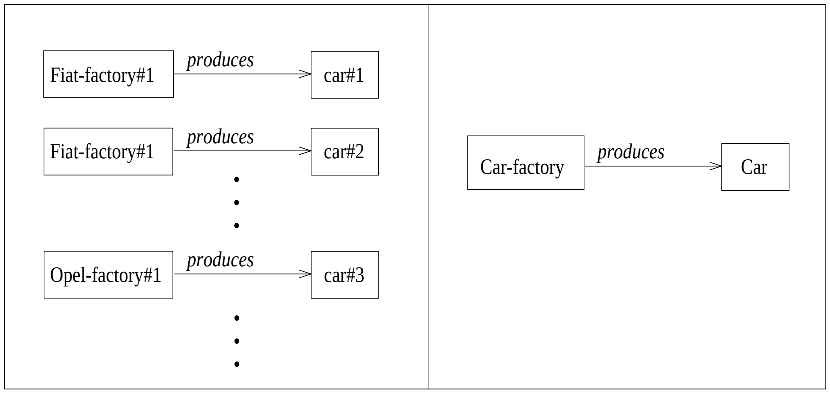

Figure 1: Examples of objects

Real world objects are distinguished with respect to their nature into *individuals* and *arrows*.

Individuals: An individual is a concrete or abstract object of independent existence.

Examples of individuals are: (i) the concrete objects *Fiat-factory#1* and *car#1*, (ii) the abstract object *Car*, that refers collectively to all cars (including *car#1*, *car#2*, *car#3*), and (iii) the abstract object *Car factory*, that refers collectively to all car factories (including *Fiat-factory#1*, *Opel-factory#1*).

Arrows: An arrow is a concrete or abstract property or binary relationship[4] from an object o to an object o'. The object o is called the $from$ object of the arrow and the object o' is called the *to* object of the arrow. In contrast to individuals, arrows do not exist independently: their existence presumes the existence of the $from$ and *to* objects. The $from$ object of an arrow a is denoted by $from(a)$ and the *to* object by $to(a)$.

Examples of arrows are: (i) the concrete relationship *Fiat-factory#1 produces car#1*, where *Fiat-factory#1* is the $from$ object and *car#1* is the *to* object, (ii) the abstract

[4]We do not make the distinction between property and binary relationship, as our approach is common to both. Thus, we use the term *arrow* to mean either a property or a binary relationship.

relationship *Car-factory produces Car*, where *Car-factory* is the $from$ object and *Car* is the *to* object. This relationship refers collectively to all concrete relationships from car companies to the cars that they produce.

In the present work, we consider only arrows whose *to* objects are individuals, while the $from$ objects can be either individuals or arrows.

The distinction of objects into individuals and arrows is based on their nature. Objects are also distinguished with respect to their concreteness into *tokens* and *classes*.

Tokens: A token is a concrete individual, such as *car#1*, or a concrete arrow, such as *Fiat-factory#1 produces car#1*.

Classes: A class is an abstract individual or an abstract arrow, in the sense that it refers collectively to a set of objects that are considered similar in some respect. Examples of classes are *Car-factory* and *Car-factory produces Car*.

Our distinction of objects into tokens and classes on one hand, and into individuals and arrows on the other, follows the structural part of the knowledge representation language Telos [28, 22].

We assume that each object is defined by a set of constraints, called the *real world intension* of the object. For a token t, the real world intension determines just the object t. For a class c, the real world intention determines the set of all objects to which c refers collectively. We call this set the *real world extension* of c and we denote it by $\mathrm{EXT}(c)$.

We assume that the real world extension of an individual class is a set of individuals. Similarly, we assume that if a is an arrow class from o to o', then both o and o' are classes and $\mathrm{EXT}(a)$ is a set of arrows from objects in $\mathrm{EXT}(o)$ to objects in $\mathrm{EXT}(o')$. Thus, the real world extension of an arrow class defines a relation[5] from $\mathrm{EXT}(o)$ to $\mathrm{EXT}(o')$.

We define two classes to be *equivalent* if they have the same real world extension. Clearly, two classes with the same real world intension are equivalent (in fact, they are equal). Roughly speaking, equivalent classes may be seen as different ways of looking at the same set of objects. In the remainder of the paper, we shall talk of classes up to equivalence.

We call *empty class*, denoted by $\perp$, the individual class whose real world extension is the empty set. Obviously, this class is unique (up to equivalence). Intuitively, the empty class corresponds to contradictory real world intensions.

[5] Throughout the paper, we use the term *relation* with its mathematical meaning.

The fragment of the real world mapped in an information base is delimited by the needs of the user and by his imperfect knowledge of the real world. The latter implies that not all objects of interest are represented in the information base. We refer to the representation of the real world in the information base, as the *model*. Though a real world object and its representation in the model are distinct, we will not differentiate between them in order to simplify our presentation.

3 The In and Isa Relations

In this section, we define the IN and ISA relations. The IN relation expresses membership of an object in the real world extension of a class, and the ISA relation expresses inclusion between real world extensions of classes.

Definition 3.1 IN relation
If an object o belongs to the real world extension of a class c then we say that o is an *instance of* c, and we denote it by $\mathrm{IN}(o, c)$. An object can be an instance of zero, one, or more classes (multiple instantiation). $\diamond$

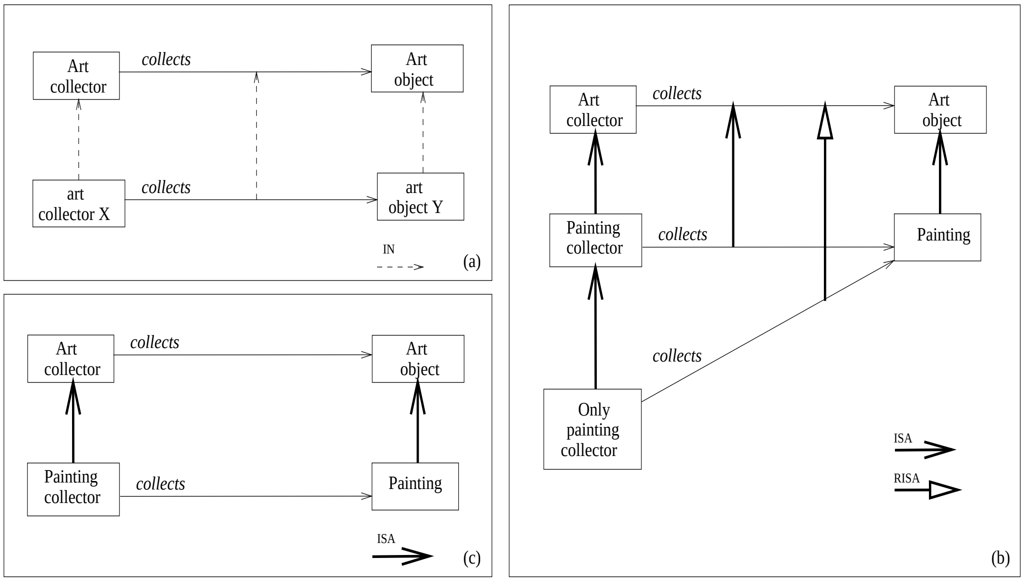

Figure 2: Example of instantiation and specialization relations between objects

For example, in Figure 2(a), the individual token *art collector X* is instance of the individual class *Art collector*. Additionally, the arrow token *collects* from *art collector X* to *art object Y* is instance of the arrow class *collects* from the individual class *Art collector* to the individual class *Art object*.

As we mentioned earlier, not all real world relations are represented in the model. We denote by In, the subset of the real world relation IN that is represented in the model. It follows that if $In(o,c)$ holds then $\mathrm{IN}(o,c)$ holds, whereas the converse is not always true.

Definition 3.2 ISA relation

Let o, o' be two classes. We distinguish two cases:

Case 1: o and o' are individual classes.

We say that o is *subclass* of o', denoted by $\mathrm{ISA}(o,o')$, if it holds that: for any individual x, if $\mathrm{IN}(x,o)$ then $\mathrm{IN}(x,o')$.

Case 2: o and o' are arrow classes.

We say that o is *subclass* of o', denoted by $\mathrm{ISA}(o,o')$, if it holds that

(i) $\mathrm{ISA}(from(o), from(o'))$, (ii) $\mathrm{ISA}(to(o), to(o'))$, and (iii) for any arrow x, if $\mathrm{IN}(x,o)$ then $\mathrm{IN}(x,o')$.

In all other cases, ISA is undefined.

A class can be subclass of zero, one, or more classes (multiple specialization). $\diamond$

For an example of ISA relation, refer to Figure 2(b), where the individual class *Painting collector* refers to art collectors that collect paintings. The arrow class *collects* from *Painting collector* to *Painting* is subclass of the arrow class *collects* from *Art collector* to *Art object* (meaning that every painting collected by a painting collector is an art object collected by an art collector). In other words, if x is an instance of the arrow *collects* of *Painting collector* then x is also an instance of the arrow *collects* of *Art collector*.

It is easy to see that the ISA relation is reflexive and transitive. However, ISA is not antisymmetric. This is because, if $\mathrm{ISA}(c,c')$ and $\mathrm{ISA}(c',c)$ hold, for classes c and c', then c and c' have the same real world extension but not necessarily the same real world intension. It follows easily, that if $\mathrm{ISA}(c,c')$ and $\mathrm{ISA}(c',c)$ holds then c and c' are equivalent. So, the ISA relation defines a partial order over classes (up to equivalence).

We denote by Isa, the subset of the real world ISA relation that is represented in the model. It follows that if $Isa(c,c')$ holds then $\mathrm{ISA}(c,c')$ holds, whereas the converse is not always true.

We introduce a number of inference rules in the model, in order to reflect properties of real world relations. For example, ISA is transitive. Thus, we introduce an inference rule that reflects ISA transitivity:

$$\text{for all classes } c_1, c_2, c_3, \quad Isa(c_1, c_2) \wedge Isa(c_2, c_3) \Rightarrow Isa(c_1, c_3).$$

This rule will allow us to *derive* $Isa(c_1, c_3)$ from $Isa(c_1, c_2)$ and $Isa(c_2, c_3)$.

The inference rules regarding *Isa* relations are called *Isa Rules* and are given in the Appendix.

4 The Restriction Isa Relation

Restriction ISA is a stronger form of ISA which is defined for arrow classes and expresses property value refinement. In what follows, we motivate its introduction.

A class can be seen as a container of information common to its instances, and each instance acquires a number of properties by class membership [29]. However, problems may arise, as several *different* semantics are possible for the acquired properties. For example, in Figure 2(c), an instance o of *Painting collector* acquires the property *collects* of *Painting collector* by being an instance of the class. However, as o is also an instance of *Art collector*, it also acquires the property *collects* of *Art collector*. This leads to two possible interpretations of *Painting collector*:

Option 1: A painting collector collects only paintings.

Option 2: A painting collector collects paintings but can also collect art objects other than paintings.

Note that in Option 1, the property *collects* of *Painting collector* "refines" the property *collects* inherited from *Art collector*. "Refinement" here is interpreted as value restriction with respect to *collects* of *Art collector* and is expressed by the Restriction ISA relation, defined below.

Definition 4.1 RISA relation

Let a, a' be two arrow classes. We say that a is a *restriction subclass* of a', denoted by RISA(a, a'), if the following hold:

(i) ISA(a, a'), and

(ii) for any arrow x, if IN(x, a') and IN$(from(x), from(a))$ then IN(x, a). $\diamond$

The specialization relations Isa and $Risa$ can be used to differentiate between possible class semantics. For an example, refer to Figure 2 (b). The declaration that *collects* of *Only painting collector* is restriction subclass of *collects* of *Art collector*, indicates that the semantics of *Only painting collector* is as in $Option$ 1. On the other hand, the declaration that *collects* of *Painting collector* is just subclass (and not restriction subclass) of *collects* of *Art collector*, indicates that the semantics of *Painting collector* is as in $Option$ 2.

We denote by $Risa$, the subset of the RISA relation that is represented in the model. It follows that if $Risa(c, c')$ holds then $\text{RISA}(c, c')$ holds, whereas the converse is not always true. Inference rules regarding $Risa$ relations are called *Risa Rules* and are given in the Appendix.

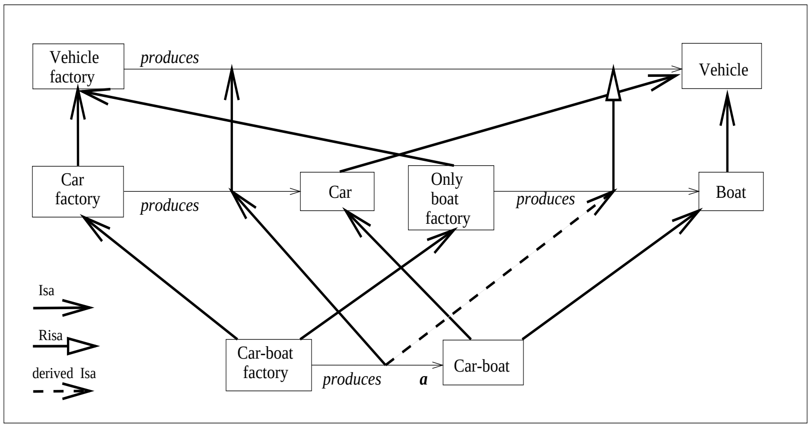

Figure 3: Example of the $Risa$ Rules

To demonstrate the application of $Risa$ Rules refer to Figure 3. Suppose that the following declarations have been made explicitly by the user:
(i) the arrow $produces$ of *Car factory* is subclass of the arrow $produces$ of *Vehicle factory*,
(ii) the arrow $produces$ of *Only boat factory* is restriction subclass of the arrow $produces$ of *Vehicle factory* (expressing that an only-boat-factory produces only boats),
(iii) *Car-boat factory* is subclass of both *Car factory* and *Only boat factory*, and
(iv) the arrow $produces$ of *Car-boat factory* (denoted by a) is subclass of the arrow $produces$ of *Car factory*.

Based on these declarations, we can make the following derivations: From Isa Rule 2 (Isa transitivity), it is derived that the arrow a is subclass of the arrow $produces$ of *Vehicle factory*. Then, from $Risa$ Rule 5, it is derived that the arrow a is also subclass of the arrow $produces$ of *Only boat factory*. This derived Isa relation (dashed line) is a non trivial relation that enriches the original conceptual schema.

5 Participation Arrows

An arrow class a relates instances of $from(a)$ with instances of $to(a)$. We say that an object o *participates* in a if there is instance of a whose $from$ object is o. In this section, we characterize an arrow class a with respect to the participation of the instances of $from(a)$ in a.

Let a be an arrow class and let c be a subclass of $from(a)$. We give the following definitions:

Necessary arrows The arrow a is called *necessary on* c, if every instance of c participates in a. For example, the arrow *flying* of class $Bird$ is a necessary arrow on the subclass *Sea gull.*

Possible arrows The arrow a is called *possible on* c, if every instance of c can participate in a (though, currently some instances of c may not participate in a). For example, the arrow *employment* of class $Person$ is possible on the subclass *Healthy adult.*

Inapplicable arrows The arrow a is called *inapplicable on* c, if no instance of c can participate in a. For example, the arrow *flying* of class $Bird$ is inapplicable on the subclass *Penguin.*

If an arrow a is necessary (resp. possible, inapplicable) on the *whole* class $from(a)$ then we simply call it *necessary* (resp. *possible*, *inapplicable*). We assume that an arrow a is inapplicable iff $to(a) = \perp$. Inapplicable arrows allow to express "negative" information, as we shall see shortly.

Let *Necessary*, *Possible*, and *Inapplicable*, be the classes of necessary, possible, and inapplicable arrows, respectively.

The inference rules regarding arrow participation are called *Participation Rules* and are given in the Appendix.

Clearly, if an arrow a is both necessary and possible then $from(a) = \perp$. Similarly, if the arrow a is both necessary and inapplicable, or both possible and inapplicable then $from(a) = \perp$. This is expressed by Participation Rule 1. It can be easily seen that any restriction subclass of a necessary (resp. possible) arrow is also necessary (resp. possible). This is expressed by Participation Rules 4 and 5.

To indicate that an arrow a is *Necessary*, we simply declare that a is instance of $Necessary$. Similarly, for *Possible* and *Inapplicable*.

Let a' be an arrow class. The information carried by a' is enriched by declaring subclasses of $from(a')$ on which a' is inapplicable, necessary, or possible. Let c be a subclass of $from(a')$. To declare that a' is inapplicable on c, we do the following (see Figure 4(a)):

1. Create an arrow a from c to $\perp$ (inapplicable arrow), and

2. declare a to be restriction subclass of a'.

We would like to mention here that the user does not need to declare explicitly that a is an inapplicable arrow, as this is derived from Participation Rule 2 (based on the fact that $to(a) = \perp$).

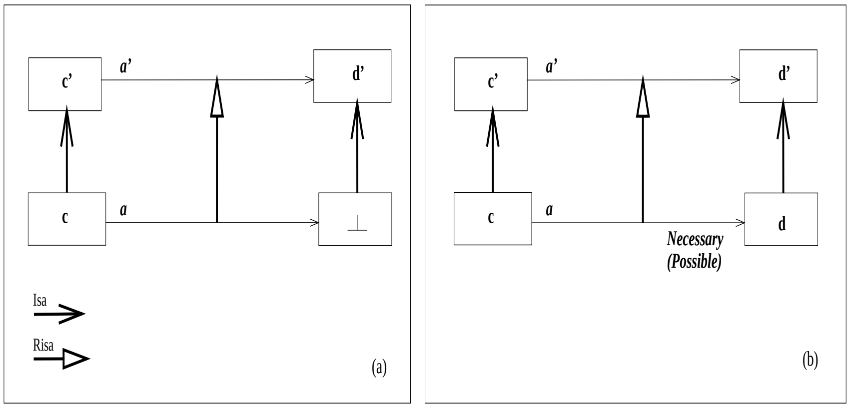

Figure 4: (a) Declaring that a' is inapplicable on c, (b) Declaring that a' is necessary (possible) on c

For example, in order to declare that the arrow *flying* of $Bird$ (denote it by a') is inapplicable on *Penguin*, we first create an arrow a from *Penguin* to $\perp$, and then declare $Risa(a, a')$.

We claim that, after declarations (1) and (2) above, a' is inapplicable on c, i.e., that there is no instance of a' whose $from$ object is instance of c. Indeed, assume x is an instance of a' whose $from$ object is instance of c. As a is a restriction subclass of a', it follows that x is an instance of a. As a is inapplicable, this leads to contradiction. It follows that a' is inapplicable on c.

Let a' be an arrow class and let c be a subclass of $from(a')$. To declare that a' is necessary (resp. possible) on c, we do the following (see Figure 4(b)):

1. Create a necessary (resp. possible) arrow a from c to a subclass d of $to(a')$[6], and

2. declare a to be subclass of a'.

We claim that, after declarations (1) and (2) above, a' is necessary (resp. possible) on c, i.e., every instance of a' whose $from$ object is instance of c must (resp. can) participate in a'. For the proof, consider first the case of necessary arrows. Let o be an instance of c. As a is a necessary arrow, o must participate in a. Therefore, there must exist an instance x of a whose $from$ object is o. As a is a subclass of a', it follows that x is an instance of a'. Therefore, a' is necessary on c. A similar proof can be given for the case of possible arrows.

In some object-oriented data models, such as EXPRESS [21], every arrow of a class is supposed to be "meaningful", for each instance of the class. In terms of our data model, this means that every arrow of a class must be necessary or possible. For example, the declaration of the arrow *maiden-name* is not allowed on the class *Person*, as it is inapplicable on male persons. Therefore, in order to declare *maiden-name*, one must first create the subclass *Female married person* and then declare *maiden-name* on *Female married person*. As the percentage of persons that cannot participate in *maiden-name* is comparable to the percentage of persons that can participate in *maiden-name*, this approach may be reasonable.

However, in situations where the percentage of class instances that cannot participate in a given arrow is very small, this approach may not be appropriate. For example, when we think of birds, we think of *flying* as being their basic property. This is so because *flying* is true for all birds, but for a few exceptions, such as penguins. Moreover, when we think of penguins, we think of *non-flying* as being their basic property. Therefore, in this example, the natural approach would be to create the arrow *flying* on the class *Bird*, and to explicitly declare that *flying* is inapplicable on *Penguin*.

Following the approach of EXPRESS, as the arrow *flying* is not allowed on the class *Bird*, one must create the subclass *Flying bird* and then create the arrow *flying* on *Flying bird* (see Figure 5(a)). There are however, two problems with this approach: (i) we are forced to create the subclass *Flying bird*, which is almost the same as the class *Bird*, just

[6] Note that d can be either an existing class or a newly declared class.

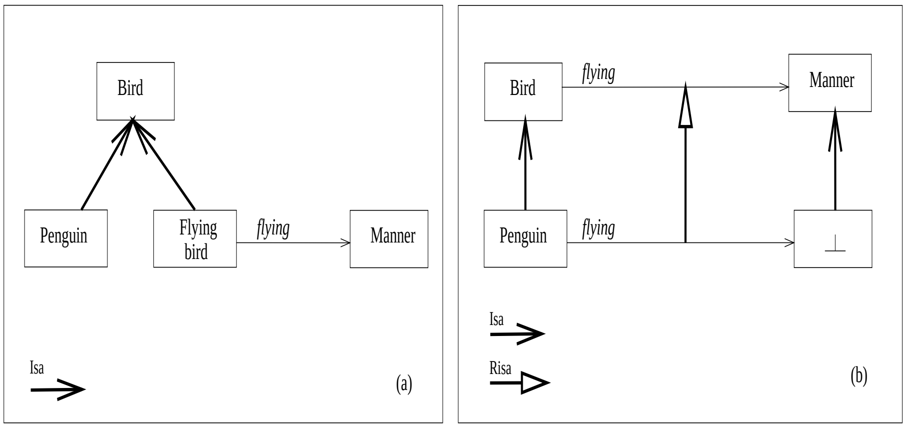

Figure 5: (a) Declaring that penguins do not fly, in EXPRESS, (b) Declaring that penguins do not fly, in our data model

to be able to declare the arrow *flying*, and (ii) the information that penguins do not fly is *implicit* in the fact that the class *Penguin* is not subclass of *Flying bird*.

Following our approach, we can overcome these problems by (i) creating the arrow *flying* on *Bird*, (ii) creating the subclass *Penguin*, which is of interest, and (iii) declaring that the arrow *flying* of *Bird* is inapplicable on *Penguin* (see Figure 5(b)). It is important to note that our approach supports both modeling styles, and thus allows us to choose the most appropriate one depending on the particular case.

6 Arrow Inheritance

Let a' be an arrow class from a class c' to a class d'. Let c be a subclass of c'. Our purpose is to define the arrow inherited by c from a', denoted by $inh(c, a')$.

The arrow a' expresses a property from c' to d' that refers collectively to a set, say E', of arrows. The $from$ objects of arrows in E' are instances of c', and their to objects are instances of d'. The question is what part of a' expresses a property of c. It is this part that we shall call the arrow inherited by c from a'. Obviously, this part is the set E of arrows in E' whose $from$ objects are instances of c. Having determined the real world extension E of the inherited arrow, it remains to determine its to object. Let $d(c, a')$ be the class of to objects of the arrows in E. We take, $d(c, a')$ to be the to object of the inherited arrow.

Following this reasoning, we define the inherited arrows as follows:

Definition 6.1 Inherited arrow

Let a' be an arrow class from a class c' to a class d' and let c be a subclass of c'. Let $d(c, a')$ be the class whose real world extension consists of the *to* objects of the arrows x such that $\mathrm{IN}(from(x), c)$ and $\mathrm{IN}(x, a')$.

We define $inh(c, a')$ to be the arrow class from c to $d(c, a')$ whose real world extension consists of the arrows x such that $\mathrm{IN}(from(x), c)$ and $\mathrm{IN}(x, a')$. The arrow $inh(c, a')$ is called the *arrow inherited* by c from a'. $\diamond$

An immediate consequence of this definition is that the inherited arrow $inh(c, a')$ is restriction subclass of the original arrow a'. The inference rules regarding inherited arrows are called *Inheritance Rules* and are given in the Appendix. In fact, Inheritance Rule 2 expresses that $inh(c, a')$ is restriction subclass of a'.

We would like to emphasize that we do *not* know what exactly the class $d(c, a')$ is. What we do know are Isa and $Risa$ relations of a' with other arrows, as well as, Isa relations of c with other classes (as declared by the user). Based on these relations and using inference rules, we can derive Isa relations of $inh(c, a')$ to other arrows. For example, if we derive that $inh(c, a')$ is subclass of an arrow a, then we can derive that $d(c, a')$ is subclass of $to(a)$ (Inheritance Rule 4). In this way, although we do not know what exactly the class $d(c, a')$ is, we can derive a set of its superclasses. Intuitively, this set provides an "approximation" of the class $d(c, a')$.

To illustrate this "approximation" process, refer to Figure 6. Applying Inheritance Rule 2, we can derive that $inh(c, a')$ is restriction subclass of a'. As a' is subclass of a_1, it is derived from Isa Rule 2 that $inh(c, a')$ is subclass of a_1. On the other hand, arrow a_0 is restriction subclass of a_1. As c is subclass of $from(a_0)$, we have all three conditions of Inheritance Rule 3 satisfied. Thus, we derive that $inh(c, a')$ is subclass of a_0.

Using Inheritance Rule 4, and based on Isa relations of $inh(c, a')$ with other arrows, we can derive that $d(c, a')$ is subclass of several classes. For example, as $inh(c, a')$ is subclass of a', it is derived that $d(c, a')$ is subclass of *Painting*. Similarly as, $inh(c, a')$ is subclass of a_0, it is derived that $d(c, a')$ is subclass of *Expensive art object*. Thus, $d(c, a')$ is subclass of both *Painting* and *Expensive art object*.

The fact that $d(c, a')$ is subclass of several classes, results in refinement of the value domain of the arrow $inh(c, a')$. Roughly speaking, if $d(c, a')$ is subclass of the classes $c_1, ..., c_n$, then the value domain of $inh(c, a')$ is the "intersection" of $c_1, ..., c_n$. For example,

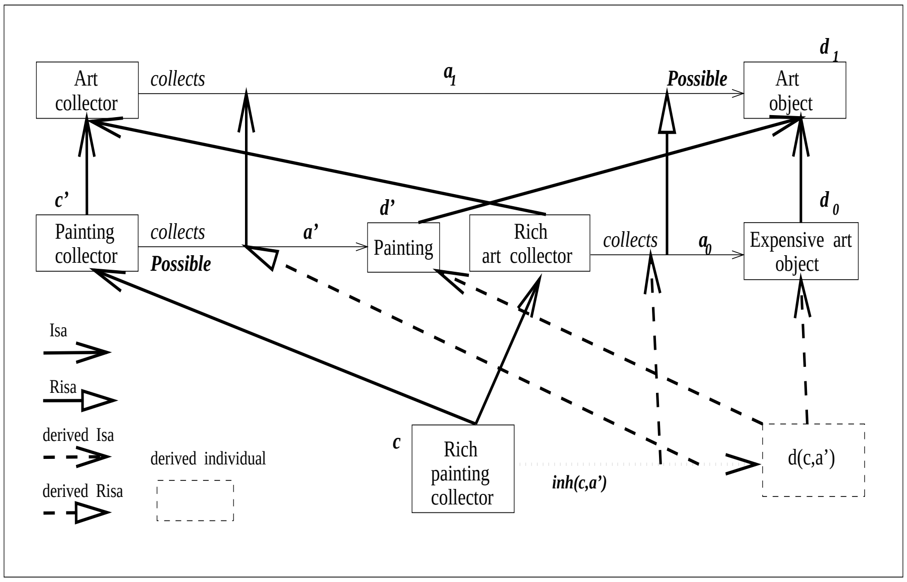

Figure 6: Example of the Inheritance Rules

in Figure 6, the value domain of a' is paintings. However, as $d(c, a')$ is subclass of *Expensive art object* and *Painting*, the value domain of $inh(c, a')$ is refined to expensive paintings (i.e., the "intersection" of *Expensive art object* and *Painting*.

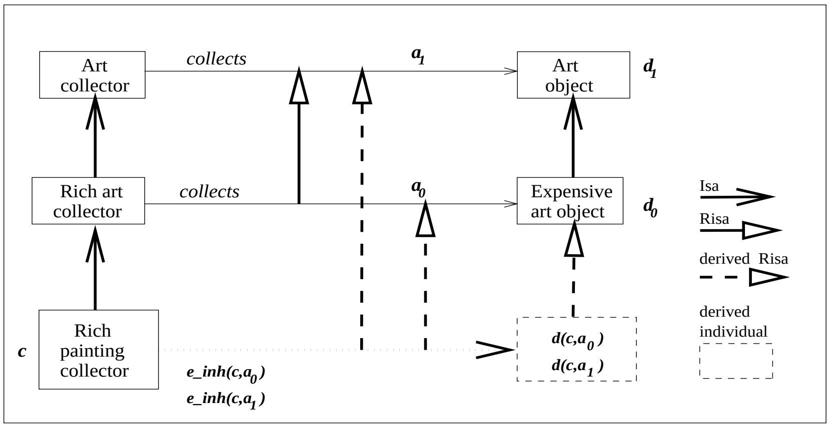

Figure 7: Example of inherited arrows that coincide

The following proposition expresses that if an arrow a_0 is restriction subclass of an arrow a_1 then the arrows inherited by a class c from a_0 and a_1 coincide (up to equivalence).

Proposition 6.1 Let a_0, a_1 be arrow classes and let c be a class. If $Risa(a_0, a_1)$ and $Isa(c, from(a_0))$ then $inh(c, a_0) = inh(c, a_1)$.

For example, in Figure 7[7], the arrows $inh(c, a_0)$ and $inh(c, a_1)$ coincide to a single derived arrow. Note that this derived arrow is restriction subclass of both a_0 and a_1. Intuitively, the arrow a_0 refines the *to* object of the arrow a_1. This refinement is expressed by the *Risa* relation. Thus, the *to* object of $inh(c, a_1)$ is, in general, the same as or finer than $to(a_0)$. This corresponds to *property refinement* (also called *type refinement*) in object-oriented data models.

Going back to the example of Figure 6, class c inherits two different arrows: one from a' (shown in the figure) and one from a_0 and a_1 (not shown in the figure)[8]. These two arrows express that a rich-painting-collector collects expensive paintings (arrow inherited from a') but he also collects expensive art objects other than paintings (arrow inherited from a_0 and a_1).

The following proposition gives an interesting property of inherited arrows. Specifically, it expresses that if an arrow inherited by a class c from an arrow a' is restriction subclass of an arrow a, then the arrows inherited by c from a and a' coincide. Intuitively, this expresses that a and a' "agree" on c.

Proposition 6.2 Let a' be an arrow class and let c be a subclass of $from(a')$. If the arrow $inh(c, a')$ is restriction subclass of an arrow a then $inh(c, a') = inh(c, a)$.

We would like to point out that if a' is necessary (resp. possible, inapplicable) then so is the inherited arrow $inh(c, a')$. This follows from Inheritance Rule 2, Participation Rules 4 and 5, and *Isa* Rule 4. For example, in Figure 6, as a' is a possible arrow, so is $inh(c, a')$.

Inheritance, as presented in this section, implies that *every* arrow of a class is inherited by its subclasses. However, an arrow a' may be *inapplicable* on a subclass c of $from(a')$, though a' is not inapplicable. For example, as we have seen in the previous section, the arrow *flying* of the class *Bird* is inapplicable on the subclass *Penguin*, though *flying* is not inapplicable on *Bird*. In such a case, we would like the arrow inherited by c from a' to be inapplicable. Roughly speaking, we would like inheritance to be "compatible" with "inapplicability". That this is so, follows from the definition of an arrow inapplicable on a subclass and the definition of inherited arrow. In fact, based on the Inheritance Rules, we can derive that the *to* object of the arrow inherited by c (and by any of its subclasses) from

[7]Note that the explicit declarations of Figure 7 come from Figure 6.

[8]Recall that as $Risa(a_0, a_1)$, the arrows inherited from a_0 and a_1 coincide.

a' is $\perp$. Intuitively, we can say that the result of inheritance of a' on c (and on any of its subclasses) is cancelled.

7 Disjoint Classes

Two classes are called *disjoint* if they do not share common instances, i.e., if their real world extensions are disjoint.

Definition 7.1 DISJ relation

Two classes c, c' are called *disjoint*, denoted by $\mathrm{DISJ}(c, c')$, if there is no object x such that $\mathrm{IN}(x, c)$ and $\mathrm{IN}(x, c')$. $\diamond$

We denote by $Disj$, the subset of the DISJ relation that is represented in the model. It follows that if $Disj(c, c')$ holds then $\mathrm{DISJ}(c, c')$ holds, whereas the converse is not always true. The inference rules regarding the $Disj$ relation are called *Disj Rules* and are given in the Appendix.

Based on $Disj$ Rules and other inference rules, we may derive that an inherited arrow is inapplicable. We distinguish two cases:

Case 1: The $from$ object of the inherited arrow has been declared to be a class other than $\perp$, whereas through Participation Rule 1, it is derived to be $\perp$. This implies that contradictory declarations have been made by the user (and an error message should be given to the user).

Case 2: The $from$ object of the inherited arrow is not derived to be $\perp$. This implies that the inherited arrow either carries valid negative information, or indicates a possible error (and a warning should be given to the user).

In what follows, we illustrate these two cases through examples. For an example of Case 1, refer to Figure 8. Assume that the user has declared $Disj(Painting, Sculpture)$. Similarly to the example of Figure 6, it is derived from the Inheritance Rules that $d(c, a')$ is subclass of both $Painting$ and $Sculpture$. From $Disj$ Rule 1, it is now derived that the classes $d(c, a')$ and $Sculpture$ are disjoint. Using this result and the fact that $d(c, a')$ is subclass of $Sculpture$, it is derived from $Disj$ Rule 2 that $d(c, a') = \perp$, i.e., the *to* object of $inh(c, a')$ is the empty class. Therefore, it is derived from Participation Rule 2, that the

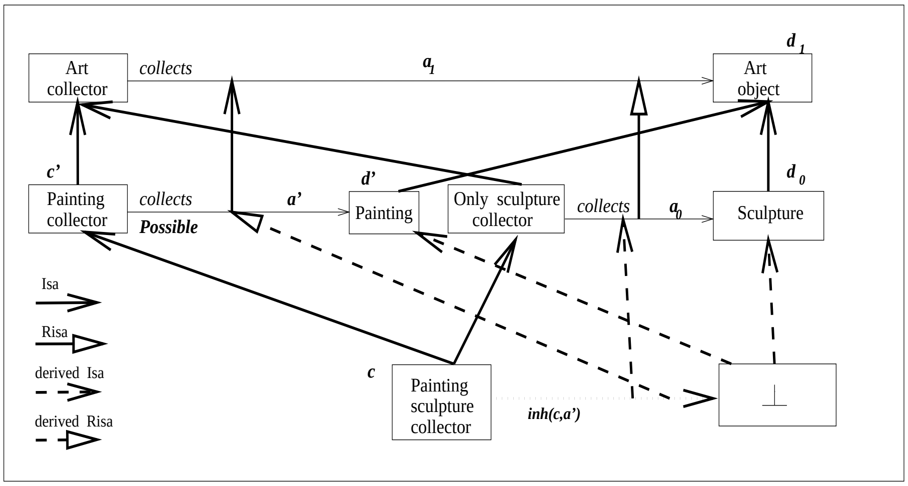

Figure 8: Inapplicable inherited arrow indicating contradictory declarations

arrow $inh(c, a')$ is inapplicable. As a' is a possible arrow, it is derived from Participation Rule 5, that $inh(c, a')$ is a possible arrow. As $inh(c, a')$ is both possible and inapplicable, it is derived from Participation Rule 1 that $c = \perp$.

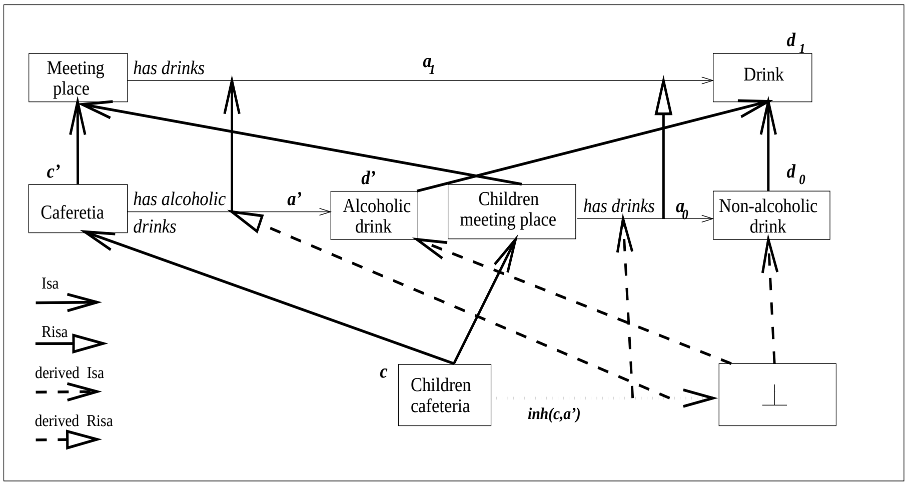

Figure 9: Inapplicable inherited arrow expressing valid negative information

For an example of Case 2, refer to Figure 9. The relation $Risa(a_0, a_1)$ expresses that all drinks served in a children meeting place should be non-alcoholic. As *Disj(Alcoholic drink, Non-alcoholic drink)*, it is derived that $d(c, a') = \perp$. This expresses the valid negative information that no alcoholic drinks are served in a children cafeteria. However, we would like to point out that an inapplicable inherited arrow does not always express valid negative information. For example, in Figure 9, assume that a' is a possible arrow but the user has neglected to declare it. Then, as in the example of Case 1, the inapplicable inherited arrow does not carry valid negative information but, rather, points-out to contradictory declarations.

8 Related Work

First, we discuss participation constraints. In our discussion, we consider two classes c and d, and an arrow class a from c to d. We also need the following definitions:
(i) c is *total* in a, if every instance of c must have an a-value in the database, and
(ii) c is *optional* in a, if not every instance of c must have an a-value in the database.

Many data models can express that c is total in a, following different approaches. For example, [19, 39, 16, 28] use a keyword (such as, *not-null*, or *necessary*) to denote that c is total in a. Other data models [27, 21], use a keyword (such as, *optional*) to denote that c is optional in a; these models interpret the absence of the keyword as "c is total in a". Finally, in several data models [35, 6, 34, 26], it is possible to specify the minimum number min of a-values that an instance of c can have. If $min > 1$ then c is total in a, otherwise, c is optional in a.

In all these data models, the concepts of *total* and *optional* refer to the database (and not to the real world). However, it is possible that an instance of c has an a-value in the real world although this value is missing from the database. In our data model, the concepts of *necessary*, *possible*, or *inapplicable* refer to the real world. Although, the fact that c is total in a implies that a is necessary, the fact that c is optional in a, gives no information on the semantics of a. It only expresses that missing a-values are allowed in the database. For example, it is possible that c is optional in a, although a may be necessary, possible, or inapplicable. Thus, we have no information as to the nature of the missing a-values. In contrast, our characterizations of properties as necessary, possible, or inapplicable, not only give information about the semantics of a but also provide a refinement of missing values into: *exists but unknown*, *can exist*, or *cannot exist*, as explained in the Introduction.

Regarding disjointness, the information that two entity classes are disjoint is represented in many data models [39, 35, 21, 26]. However, the information that two properties are disjoint is represented in very few data models, e.g., NIAM [39]. In our data model, we support both kinds of disjointness, i.e., disjointness of entity classes and disjointness of properties.

Regarding Isa and $Risa$, all data models support Isa on entity classes. However, only few data models, such as, Telos [28], NIAM [39] and OSAM* [35], support Isa between properties. On the other hand, most data models support $Risa$ implicitly based on property names. Specifically, let a class c be subclass of a class c'. If two properties a, a' of classes c and c' have the same name then $Risa(a, a')$ is implied. In our opinion, the $Risa$ relation between a and a' should be expressed explicitly by declaring $Risa(a, a')$, and not implicitly by using the same name for a and a'. Following this reasoning, we allow a and a' to have the same name even if they are not related through $Risa$. We would like to emphasize that none of the above data models, supports derivations of Isa and $Risa$ relations between properties, as is the case in our data model.

Notable shortcomings of the above mentioned data models are the following: (i) Inherited properties are not defined formally, (ii) Isa and $Risa$ relations do not interact to refine the value domain of the inherited properties, and (iii) disjoint classes do not contribute to the derivation of inapplicable inherited arrows.

The form of inheritance supported by our data model is monotonic (as any new declaration can only refine the value domain of an inherited property, but cannot "override" that property). Monotonic inheritance is supported by many data models, in particular, by terminological languages [9, 8, 31, 25, 4] and by several deductive object-oriented models (such as F-logic [23], DOT [37, 38], and QUIXOTE [40]). In what follows, we pay particular attention to terminological languages as they support many of the features of our data model.

Terminological languages support taxonomic representation of concepts. Concepts correspond to entity classes that are defined *intensionally* in terms of other concepts and necessary and sufficient conditions on their properties. These languages express that a property a of a class c is necessary, by defining c as a specialization of either the concept (SOME a)[9] or the concept (ATLEAST n a)[10], where $n \geq 1$. Additionally, they express that a property a of a class c is inapplicable, by defining c as a specialization of the concept (ALL a NOTHING)[11] or the concept (NOT (SOME a)). In terminological languages, a

[9]The concept (SOME a) refers to the set of objects that participate in a.

[10]The concept (ATLEAST n a) refers to the set of objects with at least n a-values.

[11]The concept (ALL a NOTHING) refers to the set of objects that cannot have an a-value.

concept c is declared as disjoint from concept c' either by declaring that c is specialization of the concept (NOT c') or by declaring (DISJOINT c c').

Terminological languages provide an inference mechanism for deriving inherited information. Unlike our data model, a *Risa* relation between two properties is implied by the fact that both properties have the same name. Like our data model, *Risa* relations between properties refine the value domain of the inherited properties. Additionally, disjoint classes contribute to the derivation of inapplicable inherited arrows and to the detection of contradictory schema declarations. Yet, *Isa* and *Risa* relations do not interact. For example, in Figure 6, terminological languages will not derive that the value of the property *collects* inherited by *Rich painting collector* from *Painting collector* is a specialization of *Expensive art object*. Additionally, in Figure 8, they will not derive that the property *collects* inherited by *Painting-sculpture collector* is inapplicable. Thus, the contradictory declarations of Figure 8 will not be detected.

In terminological languages, there is no uniform treatment of concepts and their instances. Thus, in contrast to our data model, terminological languages do not support uniform querying at instance and schema level, and do not support meta-concepts. We consider these to be severe limitations of terminological languages, as adding meaning to the data should be accompanied by convenient ways of querying the schema. In addition, there is no uniform treatment of concepts and their properties. Thus, in contrast to our data model, properties cannot have properties of their own.

As we mentioned earlier, F-logic [23], DOT [37, 38], and QUIXOTE [40] support monotonic inheritance. However, these data models do not consider *Isa* relations between properties. Additionally, disjointness and participation constraints are not considered.

Though the negative information expressed by inapplicable arrows is a special form of exception, our data model cannot support exceptions in general. However, in particular applications, the introduction of default, generic exceptions, and incremental modifications is useful. For example, consider the class *Patient* and its subclass *Alcoholic*. Assume that *Patient* has a property *doctor* with value domain *Physician*, and that *Alcoholic* has a property *doctor* with value domain *Psychologist*. Note that *Psychologist* is not subclass of *Physician*, as in general a psychologist is not a physician. We woule like the property *doctor* of *Alcoholic* to be an exception of the property *doctor* of *Patient*. In other words, we would like the property *doctor* of *Patient* not to be inherited by *Alcoholic*. This cannot be achieved in our data model, as the properties *doctor* of *Patient* and *doctor* of *Alcoholic* cannot be related with the *Risa* relation. What is actually needed here is non-monotonic inheritance, i.e., a form of inheritance that allows new declarations to "override" previously derived information. We mention that several approaches to non-monotonic inheritance have been proposed in the literature [7, 18, 33, 1, 23, 3][12].

Clearly, both forms of inheritance are needed, as the appropriate form depends on the particular case. For example, (i) let class *Only car factory* have a property *produces* with value domain *Car*, (ii) let class *Only boat factory* have a property *produces* with value domain *Boat*, and (iii) let class *Only car-boat factory* be subclass of both *Only car factory* and *Only boat factory*. Assume that we would like the class *Only car-boat factory* to inherit a property *produces* whose value domain is subclass of both *Car* and *Boat*. If we use non-monotonic inheritance, then the value domain of the inherited property *produces* of *Only car-boat factory* is considered to be unknown. This is because the classes *Only car factory* and *Only boat factory* are not related by *Isa*, so neither of the properties *produces* of *Only car factory* and *produces* of *Only boat factory* can "override" the other. Therefore, in this example, monotonic inheritance is the appropriate one to use.

9 Conclusions

In this paper, we have presented (i) structures that carry expressive and useful information on the database schema, (ii) a set of inference rules for schema derivations, and (iii) a mechanism for discovering contradictory schema declarations. Our basic framework is an object-oriented data model, inspired by the Telos language [28], that supports multiple instantiation, multiple classification, meta-classes, and uniform treatment of entities and their properties.

Our data model supports two forms of specialization between properties: the usual *Isa* relation, and a stronger form called *Risa*. We have demonstrated that the relations *Isa* and *Risa* can interact, in various contexts, to produce useful information regarding the semantics of classes.

Our model formalizes inheritance based on the *Risa* relation. Through inference rules,

[12][18, 33, 3] do not present an object-oriented framework. However, they consider partial ordering of declarations which can express defaults and exceptions in specialization hierarchies.

inherited properties are related to other properties with Isa and $Risa$ relations. These relations refine the value domain of the inherited properties and can be used to derive that an inherited property is necessary, possible, or inapplicable. Such derivations produce useful information about the semantics of the class inheriting the property. In addition, characterizations of an inherited property as necessary, possible, or inapplicable may uncover contradictory declarations or design errors.

We have demonstrated that the $Risa$ relation can be used by the user to declare negative information, therefore supporting a particular form of exceptions. In addition, disjointness information, in conjunction with Isa and $Risa$, can lead to the derivation of negative information (expressed in the form of inapplicable properties).

We are currently concerned with the definition of new structures that carry expressive information on the schema, such as, structures that express class membership through sufficient conditions on property values of objects. In addition, we investigate inheritance from classes to instances through class membership (called, *instance inheritance*). Though instance inheritance from classes to tokens presents no particular problem, instance inheritance from meta-classes to classes is not a trivial matter.

References

[1] S. Abiteboul, G. Lausen, H. Uphoff, E. Waller, Methods and Rules, *Proc. of the ACM SIGMOD Conference on the Management of Data*, 32-41 (1993).

[2] R. Agrawal, N.H. Gehani, Ode (Object Database and Environment): The Language and the Data Model, *Proc. of the ACM SIGMOD Intern. Conference on the Management of Data*, 36-45 (1989).

[3] A. Analyti, S. Pramanik, Reliable Semantics for Extended Logic Programs with Rule Prioritization, *Journal of Logic and Computation*, 5(3), 303-324 (1995).

[4] F. Baader, B. Hollunder, KRIS: Knowledge Representation and Inference System, *SIGART Bulletin*, 2(3), 8-14 (1991).

[5] J. Banerjee, H. Chou, J.F. Garza, W. Kim, D. Woelk, N. Ballou, H. Kim, Data Model Issues for Object-Oriented Applications, *ACM Transactions on Office Information Systems*, 5(1), 3-26 (1987).

[6] C. Batini, S. Ceri, S.B. Navathe, Conceptual and Logical Database Design: The Entity-Relationship Approach, Benjamin/Cummings, 1992.

[7] S. Brass, U.W. Lipeck, *Semantics of Inheritance in Logical Object Specifications*, C. Delobel, M.Kifer, Y. Masunaga (eds.), Proc. of the 2nd Intern. Conference on Deductive and Object-Oriented Databases, 1991, 411-430 (1991).

[8] R.J. Brachman, V.P. Gilbert, H.J. Levesque, An essential hybrid reasoning system: Knowledge and symbol level accounts of KRYPTON, *Proceedings of the 9th Intern. Joint Conference on Artificial Intelligence*, 532-539 (1985).

[9] R.J. Brachman, J.G. Schmolze, An Overview of the KL-ONE Knowledge Representation System, *Cognitive Science*, 9(2), 171-216 (1985).

[10] M.J. Carey, D.J. DeWitt, S.L. Vandenberg, A Data Model and Query Language for EXODUS, *Proc. of the ACM SIGMOD Conference on the Management of Data*, 413-423 (1988).

[11] *Communications of the ACM*, Special Issue on Next Generation Database Systems, 34(10) (1991).

[12] P. Constantopoulos, M. Doerr, The Semantic Index System: A brief presentation, *TR Institute of Computer Science Foundation for Research and Technology-Hellas*, (1994).
Available from http://www.ics.forth.gr/proj/isst/Systems/sis/.

[13] P. Constantopoulos, M. Doerr, Component Classification in the Software Information Base, O.Nierstrasz and D.Tsichritzis (eds.), *Object-Oriented Software Composition*, Prentice-Hall (1995).

[14] P. Constantopoulos, M. Theodorakis, Y.Tzitzikas, Developing Hypermedia Over an Information Repository, *Proc. of the 2nd Workshop on Open Hypermedia Systems at Hypertext'96*, (1996).

[15] P. Constantopoulos, Y.Tzitzikas, Context-Driven Information Base Update, *Proc. of the 8th Intern. Conference on Advanced Information Systems Engineering (CAiSE'96)*, 319-344 (1996).

[16] C.J. Date, An Guide to the SQL Standard, Second Edition, Addis on-Wesley, Reading, MA (1990).

[17] O. Deux et al. The story of O_2, *IEEE Transactions on Knowledge and Data Engineering*, 2(1), 91-108 (1990).

[18] D. Gabbay, E. Laenens, D. Vermeir, Credulous vs. Sceptical Semantics for Ordered Logic Programs, J. Allen, R. Fikes, E. Sandewall (eds.), *Proc. of the 2nd Intern. Conference on Knowledge Representation and Reasoning (KR'91)*, Morgan Kaufmann, 208-217 (1991).

[19] M. Hammer, D. McLeod, Database Description with SDM: A Semantic Database Model, *ACM Transactions on Database Systems*, 6(3), 351-386 (1981).

[20] R. Hull. R. King, Semantic Database Modelling, *ACM Computing Surveys*, 19(3), 202-260 (1987).

[21] ISO DIS 10303-11, Product Data Representation and Exchange - Part 11: The EXPRESS Language Reference Manual, TC184/SC4 N151, (1992).

[22] M. Jarke, S. Eherer, R. Gallersdorfer, M. A. Jeusfeld, M. Staudt, ConceptBase - A Deductive Object Base Manager, *Journal of Intelligent Information Systems, Special Issue on Advances in Deductive Object-Oriented Databases*, 4(2), 167-192 (1995).

[23] M. Kifer, G. Lausen, J. Wu, Logical Foundations of Object-Oriented and Frame-Based Languages, *Journal of the Association for Computing Machinery*, 42(4), 741-843 (1995).

[24] W. Kim, Object-Oriented Databases:Definition and Research Directions, *IEEE Transactions on Knowledge and Data Engineering*, 2(3), 327-341 (1990).

[25] A. Kobsa, First experiences with the SB-ONE knowledge representation workbench in natural-language applications, *SIGART Bulletin*, 2(3), 70-76 (1991).

[26] N. Kraicm, F. Gargouri, F. Boufares, From Object-Oriented Design Towards Object-Oriented Programming, *Proc. of the 5th Intern. Conference on Advanced Information Systems Engineering (CAiSE'96)*, 417-431 (1993).

[27] M.E.S. Loomis, Data Modeling – The IDEF-1X technique, *IEEE communications* (1986).

[28] J. Mylopoulos, A. Borgida, M. Jarke, M. Koubarakis, Telos - a Language for Representing Knowledge about Information Systems, *ACM Transactions on Information Systems*, 8(4), 325-362, (1990).

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

[30] M.P. Papazoglou, Unraveling the Semantics of Conceptual Schemas, *Communications of the ACM*, 38(9), pp.80-94 (1995).

[31] P.F. Patel-Schneider, D.L. McGuinness, R.J. Brachman, L.A. Resnick, A. Borgida, The CLASSIC Knowledge Representation System: Guiding Principles and Implementation Rationale, *SIGART Bulletin*, 2(3), 108-113 (1991).

[32] J. Peckham, F. Maryanski, Semantic Data Models, *ACM Computing Surveys*, 20(3), 153-189, (1988).

[33] N. Roos, A Logic for Reasoning with Inconsistent Knowledge, *Artificial Intelligence*, Elsevier Science Publishers, 57, 69-103 (1992).

[34] S. Spaccapietra, C. Parent, ERC+: an Object Based Entity Relationship Approach, *Conceptual Modelling, Databases and CASE: An Integrated View of Information Systems Development*, P. Loucopoulos, R. Zicari (eds.), John Wiley (1992).

[35] S.Y. W. Su, V. Krishnamurthy, H. Lam, An object-oriented semantic association model (OSAM*), *AI in Industrial Engineering and Manufacturing: Theoretical Issues and Applications*, S. Kumara, et al. (eds.) (1989).

[36] M. Stonebraker, G. Kemnitz, The POSTGRES Next-Generation Database Management System, *Communications of the ACM*, 34(10), 79-92 (1991).

[37] M. Tsukamoto, S. Nishio, M. Fujio, DOT: A Term Representation using DOT Algebra for Knowledge-bases, *Proc. of the Second Intern. Conference on Deductive and Object-Oriented Databases*, 391-410 (1991).

[38] M.Tsukamoto, S. Nishio, Inheritance Reasoning by Regular Sets in Knowledge-bases with Dot Notation, *Proc. of the Fourth Intern. Conference on Deductive and Object-Oriented Databases*, 246-264 (1995).

[39] G.M.A. Verheijen, J. van Bekkum, NIAM: an Information Analysis Method, *Information Systems Design Methodologies: A Comparative Review*, T.W. Olle, et al. (eds.), 537-590 (1982).

[40] K. Yokota, H. Tsuda, Y. Morita, Specific Features of a Deductive Object-Oriented Database Language QUIXOTE, *Proc. of the Workshop on Combining Declarative and Object-Oriented Databases*, 89-99 (1993).

[41] C. Zaniolo, Database Relations with Null Values, *Journal of Computer and System Sciences*, 28(1), 142-166 (1984).

APPENDIX: Inference Rules

We denote by I, A, C, AC the sets of individuals, arrows, classes, and arrow classes, respectively.

ISA RULES

Rule 1: $\forall\, c \in C, \quad Isa(c, c)$

Rule 2: $\forall\, c_1, c_2, c_3 \in C, \quad Isa(c_1, c_2) \;\wedge\; Isa(c_2, c_3) \;\Rightarrow\; Isa(c_1, c_3)$

Rule 3: $\forall\, o \in O,\; c, c' \in C, \quad In(o, c) \;\wedge\; Isa(c, c') \;\Rightarrow\; In(o, c')$

Rule 4: $\forall\, c \in IC, \quad Isa(c, \bot) \;\Rightarrow\; c = \bot$

RISA RULES

Rule 1: $\forall\, a, a' \in AC, \quad Risa(a, a') \;\Rightarrow\; Isa(a, a')$

Rule 2: $\forall\, x \in A,\; a_1, a_2 \in AC, \quad Risa(a_1, a_2) \;\wedge\; In(x, a_2) \;\wedge\; In(from(x),\; from(a_1)) \;\Rightarrow\; In(x, a_1)$

Rule 3: $\forall\, a \in AC, \;\; Risa(a, a)$

Rule 4: $\forall\, a_1, a_2, a_3 \in AC, \quad Risa(a_1, a_2) \;\wedge\; Risa(a_2, a_3) \;\Rightarrow\; Risa(a_1, a_3)$

Rule 5: $\forall\, a_1, a_2, a_3 \in AC,$

$$Isa(a_1, a_3) \;\wedge\; Risa(a_2, a_3) \;\wedge\; Isa(from(a_1), from(a_2)) \;\wedge\; Isa(to(a_1), to(a_2)) \Rightarrow\; Isa(a_1, a_2)$$

Rule 6: $\forall\, a_1, a_2, a_3 \in AC, \quad Isa(a_1, a_2) \;\wedge\; Isa(a_2, a_3) \;\wedge\; Risa(a_1, a_3) \;\Rightarrow\; Risa(a_1, a_2)$

PARTICIPATION RULES

Rule 1: $\forall\, a \in AC,$

$$In(a, Necessary) \;\wedge\; In(a, Possible) \;\Rightarrow\; from(a) = \bot$$
$$In(a, Necessary) \;\wedge\; In(a, Inapplicable) \;\Rightarrow\; from(a) = \bot$$
$$In(a, Possible) \;\wedge\; In(a, Inapplicable) \;\Rightarrow\; from(a) = \bot$$

Rule 2: $\forall\, a \in AC, \quad to(a) = \bot \Leftrightarrow In(a, Inapplicable)$

Rule 3: $\forall\, a, a' \in AC, \quad In(a, Inapplicable) \Rightarrow Disj(a, a')$

Rule 4: $\forall\, a, a' \in AC, \quad In(a', Necessary) \;\wedge\; Risa(a, a') \;\Rightarrow\; In(a, Necessary)$

Rule 5: $\forall\, a, a' \in AC, \quad In(a', Possible) \;\wedge\; Risa(a, a') \;\Rightarrow\; In(a, Possible)$

INHERITANCE RULES

Rule 1: $\forall\, a' \in AC,\;\; c \in C,$

$$Isa(c, from(a')) \quad\Rightarrow\quad \exists\, inh(c,a') \in AC \;\wedge\; from(inh(c,a')) = c$$

Rule 2: $\forall\, a' \in AC,\;\; c \in C,\quad Risa(inh(c,a'),\; a')$

Rule 3: $\forall\, a', a_0, a_1 \in AC,\;\; c \in C,$

$$Isa(inh(c,a'), a_1) \;\wedge\; Risa(a_0, a_1) \;\wedge\; Isa(c, from(a_0)) \;\Rightarrow\; Isa(inh(c,a'),\; a_0)$$

Rule 4: $\forall\, a' \in AC,\;\; c \in C,\quad Isa(inh(c,a'),\; a) \;\Rightarrow\; Isa(to(inh(c,a')),\; to(a))$

Rule 5: $\forall\, a' \in AC,\;\; c \in C,\;\; x \in A,\quad In(x, inh(c,a')) \;\Rightarrow\; In(to(x), to(inh(c,a')))$

Rule 6: $\forall\, a', a'' \in AC,\;\; c \in C,$

$$Isa(inh(c,a'), inh(c,a'')) \;\wedge\; Isa(inh(c,a''), inh(c,a')) \;\Rightarrow\; inh(c,a') = inh(c,a'')$$

DISJ RULES

Rule 1: $\forall\, c_0, c, c' \in C,\quad Disj(c,c') \;\wedge\; Isa(c_0, c) \;\Rightarrow\; Disj(c_0, c')$

Rule 2: $\forall\, c, c' \in IC,\quad Disj(c,c') \;\wedge\; Isa(c,c') \;\Leftrightarrow\; c = \perp$

Rule 3: $\forall\, a, a' \in AC,\quad Disj(from(a), from(a')) \;\vee\; Disj(to(a), to(a')) \;\Rightarrow\; Disj(a,a')$

Rule 4: $\forall\, a, a', a'' \in AC,$

$$Disj(a,a') \;\wedge\; Risa(a,a'') \;\wedge\; Risa(a',a'') \;\wedge\; (In(a, Necessary) \;\vee\; In(a, Possible))$$
$$\Rightarrow Disj(from(a), from(a'))$$