Fundamenta Informaticae 44 (2000) 1–31
IOS Press

Deriving and Retrieving Contextual Categorical Information through Instance Inheritance

Anastasia Analyti*
Institute of Computer Science
Foundation for Research and Technology-Hellas
Iraklio, Greece
analyti@ics.forth.gr

Nicolas Spyratos†
Laboratoire de Recherche en Informatique
Universite de Paris-Sud
Orsay Cedex, France
spyratos@lri.fr

Panos Constantopoulos*
Institute of Computer Science
Foundation for Research and Technology-Hellas
Iraklio, Greece

Department of Computer Science
University of Crete
Iraklio, Greece
panos@ics.forth.gr

Abstract. In semantic and object-oriented data models, each class has one or more typing properties that associate it to other classes, and carry type information about *all* instances of the class. We introduce a new kind of property that we call *instance-typing property*. An instance-typing property associates an instance of a class to another class, and carries

*Address for correspondence: Institute of Computer Science, Foundation for Research and Technology-Hellas, Iraklio, Greece

†Research conducted while this author was visiting with the Institute of Computer Science, Foundation for Research and Technology-Hellas. Address for correspondence: Laboratoire de Recherche en Informatique, Universite de Paris-Sud, Orsay Cedex, France

type information about that particular instance (and not about all instances of the class). Instance-typing properties are important as they allow to represent *summary* information about an instance, in addition to specific information.

In this paper, we study inheritance of properties from a class to an instance, using type information about the class, as well as type information about the instance. This kind of inheritance, that we call *contextual instance-inheritance*, provides us with the most specific type information about the instance, in a particular context. Intuitively, a *context* is a metaclass of interest with respect to which this most specific information is determined.

We demonstrate that contextual instance-inheritance is a powerful conceptual modeling mechanism, capable of expressing valuable information about instances. We also provide a framework in which derived instance-inherited properties can be represented and retrieved in the same way as "usual" properties.
Keywords: instance-inheritance, type information, context, inference rules, conceptual modeling, object-oriented modeling.

1. Introduction

In semantic and object-oriented data models [8, 16, 19, 25], a class refers collectively to a set of objects, called its *instances*, that are similar in some respect. For example, in Figure 1, *Car factory* is a class that refers collectively to all particular car factories. Each class has one or more typing properties that associate it to other classes, and carry type information about *all* instances of the class. For example, *Car factory* has the typing property *produces* that associates it to the class *Car*, and expresses the type information that all car factories produce cars.

In this paper, we introduce a new kind of property that we call *instance-typing property*. An instance-typing property associates a particular instance of a class to another class, and carries type information about that particular instance (and not about all instances of the class). For example, in Figure 1, the property *produces* of *factory#2* is an instance-typing property, and carries the type information that all cars produced by *factory#2* are racing cars.

Instance-typing properties are important as they allow to declare *summary* information about an instance, in addition to specific information. Moreover, instance-typing properties allow to express *incomplete* information, in the case where specific information about the instance is missing. For example, a user may want to declare that all cars produced by *factory#2* are racing cars, though he may not know the particular cars that *factory#2* produces.

In object-oriented systems, one can query about specific information carried by the typing properties of a class. For example, one can query about the particular cars produced by a given car factory. However, querying about summary information, such as the type of cars produced by a given car factory is also important, as the user may be interested just in the type of cars, and not in the particular cars produced by a factory.

For example, refer to Figure 1, where *factory#1* is instance of *Convertible-sports-car factory*, and thus of *Convertible-car factory*. If we assume that the property *produces* of *Convertible-car factory* specializes that of *Car factory*, then we can infer that all cars produced by *factory#1* are

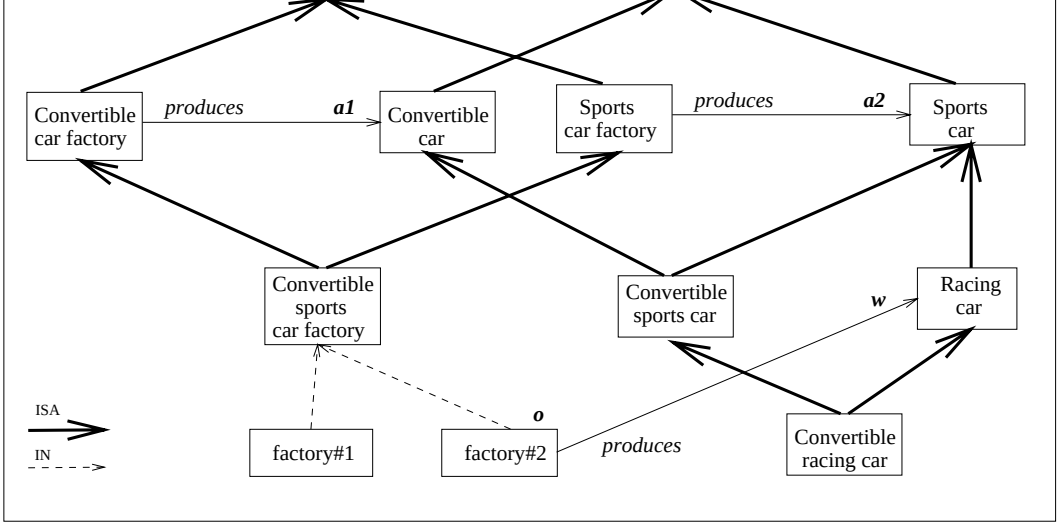

Figure 1. Example for deriving categorical information

convertible cars. Similarly, as *factory#1* is instance of *Sports-car factory*, if we assume that the property *produces* of *Sports-car factory* specializes that of *Car factory*, then we can infer that all cars produced by *factory#1* are sports cars.

Consider now the query "what are the types of cars produced by *factory#1*". An easy answer would be: *Convertible car* and *Sports car*. This answer, however, is ambiguous as we do not know whether all cars produced by *factory#1* are both convertible and sports (i.e. convertible-sports cars), or whether some are convertible and some are sports. We propose to answer this query by giving the most specific type of cars produced by *factory#1*, i.e. convertible-sports cars. We refer to this most specific type of cars as the *category* of cars produced by *factory#1*. In our model, we express this information by deriving an instance-typing property of *factory#1* with value *Convertible-sports car* (the intersection of the classes *Convertible car* and *Sports car*). We call this property, the *categorical property inherited* by *factory#1* from the property *produces* of *Car factory*.

For another example, refer again to Figure 1, where *factory#2* has the instance-typing property *produces*, giving the information that all cars it produces are racing cars. Obviously, the car category produced by *factory#2* is *Convertible-racing car* (the intersection of the classes *Convertible-sports car* and *Racing car*). Therefore, the category of cars produced by *factory#2* is *Convertible-racing car*.

So far, we have implicitly assumed that the classes *Car*, *Convertible car*, *Sports car*, *Convertible-sports car*, and *Racing car* are instances of the metaclass *Vehicle type*. Therefore, the category *Convertible-sports car* produced by *factory#1* has been in fact derived *in the context* of *Vehicle type*, and so has the category *Convertible-racing car* produced by *factory#2*.

Obviously, if we change context, then we expect to derive different categories for *factory#1* and *factory#2*. For example, assume that we are now interested in instances of the metaclass

Basic-land-vehicle type. Additionally, assume that *Car* and *Sports car* are instances of this metaclass, whereas *Convertible car*, *Convertible-sports car*, and *Racing car* are not. In this case, we can derive that the category of cars produced by *factory#1* is *Sports car* (and not *Convertible-sports car*), and the category of cars produced by *factory#2* is also *Sports car* (and not *Convertible-racing car*).

Therefore, given a context, each instance, such as *factory#1* or *factory#2*, inherits a categorical property. We propose to collect all these inherited categorical properties in the form of a property that we call *inherited multi-categorical property*. This is a novel concept that allows retrieval of inherited categorical properties, in the same way as for "usual" properties. An additional feature of an inherited multi-categorical property is that its value provides summary information about the values of the inherited categorical properties to which it refers.

Relating inherited categorical and multi-categorical properties (generally called, *instance-inherited properties*) to other properties may cause specialization of their value. To our knowledge, no existing data model supports retrieval of inherited categorical properties in a context, or the concept of inherited multi-categorical property. Support for this kind of retrieval is the main goal of our work.

Certain models, such as terminological languages [7, 5, 24, 21, 4] or *F-logic* [6], do support instance-inheritance. However, although terminological languages support instance inheritance with value specialization, they do not support reference to a context or inherited multi-categorical properties (see also subsection 8.1). As for F-logic, it supports default instance-inheritance, a kind of inheritance that does not provide for value specialization or for reference to a context (see also subsection 8.3).

We would like to emphasize that our model can be implemented in any system supporting the specification of deductive rules, such as ConceptBase [18], or F-logic [6].

In summary, the main contributions of this paper are:

1. A framework for the uniform retrieval of "usual", as well as inherited categorical properties.
2. Support for a special form of incomplete information, expressed through declared instance-typing properties, as well as through inherited categorical properties.

The rest of the paper is organized as follows: Section 2 describes our view of real world objects. Section 3 presents contextual subclass-inheritance, and Section 4 presents contextual instance-inheritance. Section 5 defines specialization by restriction and demonstrates how it can be used to differentiate between possible semantics of inheritance. Section 6 defines the instance-inheritance relations, and demonstrates their use for deriving and retrieving categorical information. Section 7 introduces a set of inference rules for deriving contextual categorical and multi-categorical information, and presents examples that illustrate the use of these rules. Section 8 discusses related work, and Section 9 contains concluding remarks and suggestions for further research.

The proofs of all propositions are given in Appendix A.

2. Our view of real world 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, such as the concrete object *car#1*, or the abstract object *Car*.

Arrows: An arrow is a concrete or abstract property or binary relationship[1] from an object o to an object o', such as *factory#1 produces car#1*, or *Car-factory produces Car*. 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 have an independent existence: their existence presumes that of the $from$ and *to* objects. The $from$ object of an arrow a is denoted by $from(a)$ and the *to* object by $to(a)$. In the present work, we consider only arrows whose $from$ and *to* objects are individuals.

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*, as follows.

Individual Tokens: An individual token is a concrete individual, e.g. *car#1* is an individual token, whereas *Car* is not.

Individual Classes: We define individual classes recursively as follows:

(i) An individual L_1-class is an abstract individual that refers collectively to a set of individual tokens, called its *instances*. For example, *Car* is an individual L_1-class, as it refers collectively to individual tokens, such as *car#1*.

(ii) An individual L_i-class, where $i > 1$, is an abstract individual that refers collectively to a set of individual L_{i-1}-classes, called its *instances*. For example, *Vehicle type* is an individual L_2-class, as it refers collectively to L_1-classes, such as *Car* or *Boat*.

(iii) An *individual class* is any individual L_i-class, where $i \geq 1$, and an *individual metaclass* is any individual L_i-class, where $i > 1$. In the rest of the paper, whenever there is no ambiguity, we use the terms class and metaclass to mean individual class and individual metaclass, respectively.

Arrow Tokens: An arrow token is one of the following:

(i) A concrete arrow, such as *factory#1 produces car#1*.

(ii) An abstract arrow a from an individual o to an individual class c, that refers collectively to a set of concrete arrows going from o to instances of c. The arrow a is called *instance-typing arrow*, and the concrete arrows are called *categorical instances* (or *c-instances*) of a. For example, *factory#1 produces Car* is an instance-typing arrow, as it refers collectively to the concrete arrows going from *factory#1* to instances of *Car*, i.e. to the particular cars that factory#1 produces.

[1] 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.

Arrow classes: We define arrow classes recursively as follows:

(i) An arrow L_1-class is an abstract arrow a from an individual class c to an individual class d that refers collectively to a set of arrow tokens whose $from$ objects are instances of c and whose to objects are instances of d. The arrow tokens are called the *instances* of a. For example, *Car-factory produces Car* is an arrow L_1-class, and *factory#1 produces car#1* is an instance of this class.

(ii) An arrow L_i-class, where $i > 1$, is an abstract arrow a from an individual class c to an individual class d that refers collectively to arrow L_{i-1}-classes going from instances of c to instances of d. These arrow L_{i-1}-classes are called *instances* of a. For example, *Vehicle-factory-type produces Vehicle-type* is an arrow L_2-class, and *Car-factory produces Car* is an instance of this class.

(iii) An *arrow class* is any arrow L_i-class, where $i \geq 1$, and an *arrow metaclass* is any arrow L_i-class, where $i > 1$.

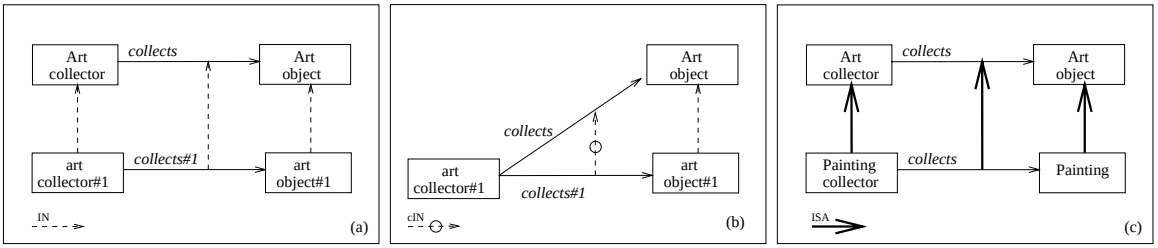

Figure 2. Examples of relations IN, cIN, and ISA

If an object o is instance of an individual (resp. arrow) class c then we write $\text{IN}(o, c)$. In our model, an individual (resp. arrow) can be instance of zero, one, or more individual (resp. arrow) classes. For example, in Figure 2(a), the arrow token *collects#1* from *art collector#1* to *art object#1* is instance of the arrow class *collects* from *Art collector* to *Art object*.

The set of instances of an individual (resp. arrow) class c is called the *extension* of c, and it is denoted by EXT(c).

If a concrete arrow x is c-instance of an instance-typing arrow y then we write $\text{cIN}(x, y)$. For example, in Figure 2(b), the arrow token *collects#1* from *art collector#1* to *art object#1* is c-instance of the instance-typing arrow *collects* from *art collector#1* to *Art object*. The set of c-instances of y is called the *categorical-extension* (or c-extension) of y.

We give now the definition of the specialization relation ISA for individual classes, as well as for arrow classes. For individual classes, ISA is just set inclusion of the corresponding extensions, while for arrow classes we must have set inclusion between the extensions of the $from$ and to objects, in addition to the inclusion between the extensions of the arrow classes.

Definition 2.1. ISA relation
Case 1: c and c' are individual classes.
We say that c is *subclass* of c', denoted by $\text{ISA}(c, c')$, iff it holds that: for any individual x, if $\text{IN}(x, c)$ then $\text{IN}(x, c')$.
Case 2: c and c' are arrow classes.
We say that c is *subclass* of c', denoted by $\text{ISA}(c, c')$, iff it holds that
(i) $\text{ISA}(from(c), from(c'))$, (ii) $\text{ISA}(to(c), to(c'))$, and (iii) for any arrow x, if $\text{IN}(x, c)$ then $\text{IN}(x, c')$.
In all other cases, ISA is undefined.
An individual (resp. arrow) class can be subclass of zero, one, or more individual (resp. arrow) classes. $\diamond$

For an example of ISA relation between arrow classes, refer to Figure 2(c). 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). It follows that, 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 anti-symmetric. We define two individual or arrow classes c and c' to be *equivalent* iff both $\text{ISA}(c, c')$ and $\text{ISA}(c', c)$ hold. So, the ISA relation defines a partial order over classes (up to equivalence). In this paper, we shall talk about classes up to equivalence.

3. Subclass Inheritance

In this section we define simple and contextual inheritance from a class to a subclass. In the following sections, we relate these concepts to contextual instance-inheritance. Simple inheritance has been introduced in earlier works [2, 3], and presented here to facilitate the definition of contextual inheritance. Contextual inheritance extends simple inheritance, forming the main concept of this section.

3.1. Simple Subclass Inheritance

Intuitively, subclass-inheritance means that every property of a class is inherited by its subclasses. For example, the property *produces* of the class *Car factory* is inherited by its subclass *Only-sports-car factory* (see Figure 3(a)). However, car factories produce any kind of car, whereas only-sports-car factories produce only sports cars. In other words, when going from the class to the subclass, not only the $from$ object of the property is specialized, but also its *to* object. This motivates the following definition of subclass-inheritance.

Definition 3.1. 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 E be the set of arrows x such that $\text{IN}(x, a')$ and $\text{IN}(from(x), c)$. Let $D(c, a')$ be the class whose extension

is the set of *to* objects of the arrows in E. We define the *arrow inherited* by c from a', denoted by $inh(c, a')$, to be the arrow class from c to $D(c, a')$ whose extension is E. $\diamond$

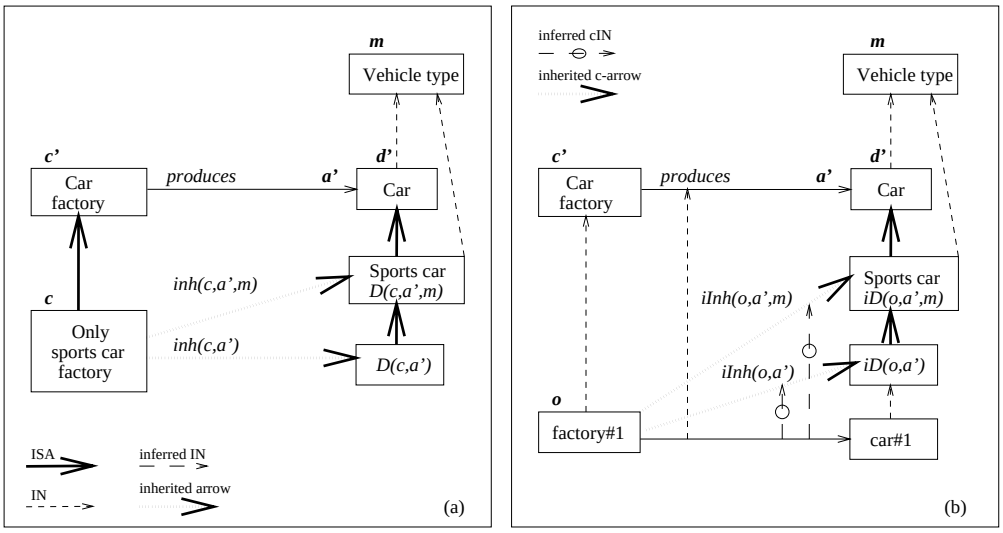

Figure 3. (a) Subclass-inheritance, (b) Categorical instance-inheritance

As the *to* object of each arrow in E is instance of d', it follows that $D(c, a')$ is subclass of d'. For an example of inherited arrow, refer to Figure 3(a), where a' is the arrow *produces* from *Car factory* to *Car*, and c is the class *Only-sports-car factory*. The extension of $inh(c, a')$ is the set of all instances of a' whose $from$ objects are instances of c. In other words, E is the set of arrows that go from only-sports-car factories to the cars they produce. It follows that the extension of $D(c, a')$ is the set of all cars produced by only-sports-car factories, and therefore $D(c, a')$ is subclass of *Sports car*.

3.2. Contextual Subclass Inheritance

So far, we have ignored the fact that d' may be instance of some metaclass m (as in Figure 3(a)). Intuitively, such a metaclass can be thought of as a universe of discourse, in which we talk about vehicle types. Now, although $D(c, a')$ is subclass of d', it may not be an instance of m. In such a case, if we want to talk about $D(c, a')$ in the universe of discourse m, then we must designate one of the instances of m as a surrogate for $D(c, a')$. For example, in Figure 3(a), if $D(c, a')$ is not an instance of m and *Sports car* is, then *Sports car* can be the surrogate for $D(c, a')$ in m.

In order for such a surrogate to make sense, it must "approximate" $D(c, a')$ as closely as possible, i.e. (i) it must be a minimal superclass of $D(c, a')$, and (ii) it must be unique. To satisfy these requirements on the surrogate, we require that the instances of m form a lower semi-lattice. We call *cover*, any surrogate that satisfies (i) and (ii) above, and we call *context* any metaclass whose instances form a lower semi-lattice. More formally, we have the following definitions.

Definition 3.2. Meet class
Let c, c' be two individual classes. The *meet* of c and c', denoted by $meet(c, c')$, is the individual class defined by: $\text{EXT}(meet(c, c')) = \text{EXT}(c) \cap \text{EXT}(c')$ $\diamond$

Recall that we talk of classes up to equivalence, so $\text{EXT}(meet(c, c'))$ is sufficient in order to define the class $meet(c, c')$. Obviously, $meet(c, c')$ is subclass of both c and c'.

Definition 3.3. Context
Let m be an individual metaclass. We say that m is a *context* iff it holds that: for any classes c, c', if $\text{IN}(c, m)$, $\text{IN}(c', m)$ then $\text{IN}(meet(c, c'), m)$. $\diamond$

It can be easily seen that, for any instances c and c' of m, $meet(c, c')$ is the greatest lower bound of c and c' with respect to ISA. Therefore, the instances of a context form a lower semi-lattice with respect to ISA.

We call *empty class*, the individual class whose extension is the empty set, and we call two instances *inconsistent* iff their meet is the empty class. Note that Definition 3.3 allows for inconsistent classes to be instances of a context. However, if a context has two inconsistent instances, then it must also have the empty class as instance.

An example of context is the metaclass *Article-subject type*, whose instances are classes of articles described by conjunctions of keywords. Indeed, if we consider any two instances of this metaclass, for example the class *Logic* and the class *Database* , then their conjunction, here *Logic & Database*, is also instance of the metaclass.

As a more abstract example of context, call S_i the individual L_i-class that refers collectively to all individual L_{i-1}-classes, for $i > 1$. Obviously, S_i is a context, as the meet of any two individual L_{i-1}-classes is also an individual L_{i-1}-class.

Definition 3.4. Cover
Let c be an individual class and let m be a context. The *cover* of c in m is the least superclass of c in the semi-lattice of instances of m, if such a superclass exists. Otherwise, the cover of c in m is undefined. $\diamond$

Intuitively, the class c can be seen as a concept (which may not belong to the universe of discourse m), and the cover of c in m as a concept of m that minimally generalizes c.

We are now ready to define arrow inheritance in the context of a metaclass.

Definition 3.5. Inherited arrow in a context
Let a' be an arrow class from a class c' to a class d', such that d' is instance of a context m. Let c be a subclass of c'. Let E be the set of arrows x such that $\text{IN}(x, a')$ and $\text{IN}(from(x), c)$. Let $D(c, a')$ be the class whose extension is the set of *to* objects of the arrows in E, and let $D(c, a', m)$ be the cover of $D(c, a')$ in m. We define the *arrow inherited* by c from a' in m, denoted by $inh(c, a', m)$, to be the arrow class from c to $D(c, a', m)$ whose extension is E. $\diamond$

As m is a context, d' is instance of m, and $D(c, a')$ is subclass d', the cover of $D(c, a')$ in m exists. Therefore, $D(c, a', m)$ is instance of m. We will show next that $D(c, a', m)$ is also subclass of d'. As both $D(c, a', m)$ and d' are instances of m, and m is a context, it follows that $meet(D(c, a', m), d')$ is instance of m. As $meet(D(c, a', m), d')$ is subclass of $D(c, a', m)$, it follows that $D(c, a', m) = meet(D(c, a', m), d')$. Therefore, $Isa(D(c, a', m), d')$ holds.

For an example, refer to Figure 3(a), where m is the context *Vehicle type*, and *Sports-car* is instance of m. As not every sports-car is produced by a sports-car factory, $D(c, a')$ is strict subclass of *Sports-car*. If *Sports-car* happens to be the cover of $D(c, a')$ in m, then the *to* object of $inh(c, a', m)$ will be *Sports-car*. Obviously, *Sports-car* should be subclass of *Car*.

We would like to emphasize here that the important concept (introduced by Definition 3.5) is contextual subclass inheritance, expressed by the arrow $inh(c, a', m)$. The definitions of $D(c, a')$ and $inh(c, a')$ have been introduced essentially in order to facilitate the definitions of $D(c, a', m)$ and $inh(c, a', m)$, respectively.

There is an interesting connection between contextual inheritance and individual metaclasses. Indeed, assume that d' is an individual L_{i-1}-class, for $i > 1$, and let S_i be the individual L_i-class that refers collectively to all individual L_{i-1}-classes. As we have seen earlier S_i is a context. Let a' be an arrow class from a class c' to d', and let c be a subclass of c'. Note that $D(c, a')$ is instance of S_i, thus the cover of $D(c, a')$ in S_i is $D(c, a')$ itself. In other words, $D(c, a', S_i) = D(c, a')$. It follows that $inh(c, a', S_i)$ coincides with $inh(c, a')$.

4. Instance Inheritance

In this section, we define contextual categorical and multi-categorical inheritance, from an instance to a class.

4.1. Contextual Categorical Inheritance

Let a' be an arrow L_1-class from a class c' to a class d' and let o be an instance of c' (as in Figure 3(b)). In order to define the categorical arrow inherited by o from a' in a context m, we proceed in a similar manner as for subclass-inheritance in the previous section.

Definition 4.1. Inherited categorical arrow
Let a' be an arrow L_1-class from a class c' to a class d'. Let object o be instance of c'. Let E be the set of arrows x such that $from(x) = o$ and $\text{IN}(x, a')$ holds. Let $iD(o, a')$ be the class whose extension is the set of *to* objects of the arrows in E. We define the *categorical arrow* (or *c-arrow*) *inherited* by o from a', denoted by $iInh(o, a')$, to be the instance-typing arrow from o to $iD(o, a')$ whose c-extension is E. $\diamond$

As the *to* object of each arrow in E is instance of d', it follows that $iD(o, a')$ is subclass of d'. For an example, refer to Figure 3(b), where o is the individual token *factory#1*. Then, $iD(o, a')$ is the class whose extension is the set of all cars produced by *factory#1*, and E is the set of all

arrows from *factory#1* to instances of $iD(o, a')$. Thus, the inherited c-arrow $iInh(o, a')$ is the instance-typing arrow from *factory#1* to the class $iD(o, a')$ whose c-extension is E.

We now extend the previous definition to the case where the *to* object of the inherited c-arrow is instance of a context m. Assume that d' is an instance of a context m, where m is a metaclass. In certain cases, we may like to view the *to* object of the c-arrow inherited by c from a' as an instance of m. Therefore, we extend the previous definition of inherited c-arrow, so that the *to* object of the inherited c-arrow is instance of m. To this end, we augment $iD(o, a')$ up to the "closest" instance of m. Obviously, this "closest" instance is the cover of $iD(o, a')$ in m.

Definition 4.2. Inherited categorical arrow in a context
Let a', c', d', E, and $iD(o, a')$ be as in Definition 4.1. Additionally, assume that d' is instance of a context m, and that $iD(o, a', m)$ is the cover of $iD(o, a')$ in m. We define the *categorical arrow* (or *c-arrow*) *inherited* by o from a' in m, denoted by $iInh(o, a', m)$, to be the instance-typing arrow from o to $iD(o, a', m)$ whose c-extension is E. $\diamond$

It holds that $iD(o, a', m)$ is instance of m. It also holds that $iD(c, a', m)$ is subclass of d'. Continuing with the example of Figure 3(b), assume that the cover of $iD(o, a')$ in m is *Sports car*. Then, the inherited c-arrow $iInh(o, a', m)$ is the instance-typing arrow from *factory#1* to *Sports car* whose c-extension is E.

4.2. Contextual Multi-categorical Inheritance

Let a'' be an arrow L_2-class from a metaclass c'' to a metaclass $'d''$ (as in Figure 4), such that d'' is a context. Let c' be an instance of c''. For every instance o of c', and for every instance y of a'', there is a c-arrow $iInh(o, y, d'')$ inherited by o from y in d''. We would like to collect all such arrows $iInh(o, y, d'')$ into a single arrow that we shall call *inherited m-arrow*. Obviously, the $from$ object of this arrow is the class c', whereas the *to* object is subclass of d''. Therefore, we define the inherited m-arrow similarly to the inherited c-arrow as follows:

Definition 4.3. Inherited multi-categorical arrow
Let a'' be an arrow L_2-class from a metaclass c'' to a metaclass d'', such that d'' is a context. Let the class c' be instance of c'', and let E be the set of inherited c-arrows $iInh(o, y, d'')$, such that $\text{IN}(o, c')$ and $\text{IN}(y, a'')$. Let $iD_2(c', a'')$ be the class whose extension is the set of *to* objects of the arrows in E. We define the *multi-categorical arrow* (or *m-arrow*) *inherited* by c' from a'', denoted by $iInh_2(c', a'')$, to be the arrow class from c' to $iD_2(c', a'')$ whose extension is E. $\diamond$

It holds that $iD_2(c', a'')$ is subclass of d''. For an example of m-arrow, refer to Figure 4, and note that (i) *factory#1* is instance of the classes *Car factory*, *Boat factory*, and *Vehicle factory*, (ii) the arrow y_1 is instance of a'', and (iii) the arrow y_2 is instance of a''. Therefore, the c-arrows inherited by *factory#1* from y_1 and y_2 in d'', are instances of $iInh_2(c', a'')$.

Assume now that d'' is instance of a context m', where m' is a metaclass (such a context is not shown in Figure 4). In certain cases, we may like to view the *to* object of the m-arrow inherited by c' from a'' as an instance of m'. Therefore, we extend the previous definition of inherited m-arrow, so that the *to* object of the inherited m-arrow is instance of m'.

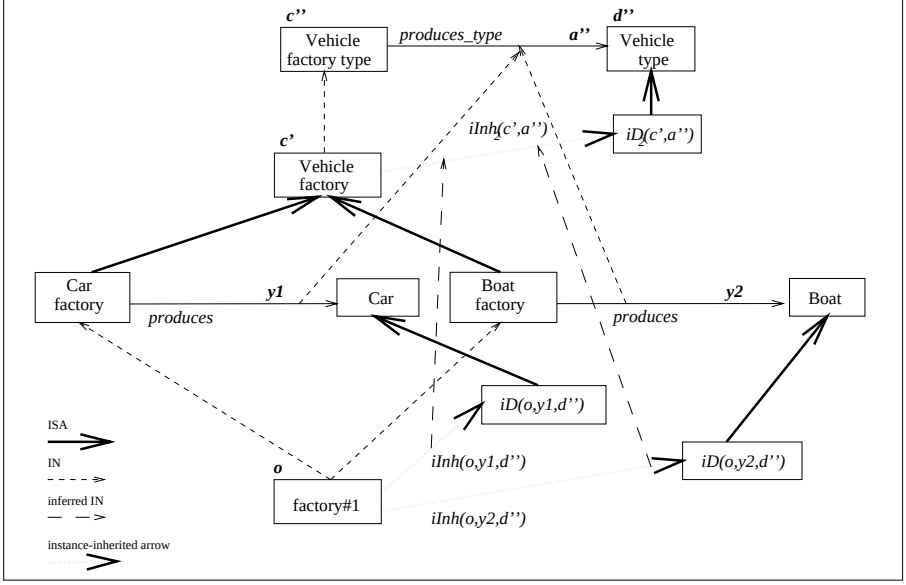

Figure 4. Example of multi-categorical instance-inheritance

Definition 4.4. Inherited multi-categorical arrow in a context
Let a'', c'', d'', c', E, and $iD_2(c', a'')$ be as in Definition 4.3. Additionally, assume that d'' is instance of a context m', and that $iD_2(c', a'', m')$ is the cover of $iD_2(c', a'')$ in m'. We define the *multi-categorical arrow* (or *m-arrow*) *inherited* by c' from a'' in m', denoted by $iInh_2(c', a'', m')$, to be the arrow class from c' to $iD_2(c', a'', m')$ whose extension is E. $\diamond$

It holds that $iD_2(c', a'', m')$ is instance of m'. Additionally, it holds that $iD_2(c', a'', m')$ is subclass of d''.

5. Specialization by Restriction

In this section, we review the RISA relation between arrow classes. The RISA relation is a stronger form of ISA, and was first introduced in [2, 3].

A class can be seen as a container of information common to its instances, and each instance "inherits" a number of properties by class membership [22]. However, problems may arise, as several *different* semantics are possible for the inherited properties. For example, in Figure 5(a), an instance o of *Painting collector* inherits the property *collects* of *Painting collector* by being an instance of this class. However, as o is also instance of *Art collector*, it also inherits the property *collects* of *Art collector*. Therefore, o has two inherited properties that may or may not coincide. This leads to two possible interpretations of *Painting collector*:

Option 1: A painting collector collects only paintings (this is the case where the two inherited properties coincide).

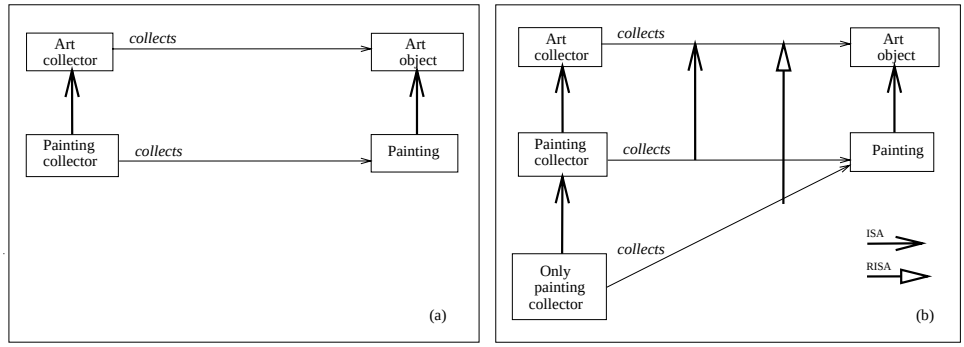

Figure 5. Introducing the RISA relation

Option 2: A painting collector collects paintings, but also art objects other than paintings (this is the case where the two inherited properties do not coincide).

Note that in Option 1, the property *collects* of *Painting collector* "specializes" the property *collects* of *Art collector*. "Specialization" here is interpreted as value restriction with respect to *collects* of *Art collector*, and is expressed by the RISA relation.

Definition 5.1. RISA relation
Let a, a' be two arrow classes. We say that a is a *restriction subclass* of a', denoted by $\mathrm{RISA}(a, a')$, iff the following hold:
(i) $\mathrm{ISA}(a, a')$, and
(ii) for any arrow x, if $\mathrm{IN}(x, a')$ and $\mathrm{IN}(from(x), from(a))$ then $\mathrm{IN}(x, a)$. $\diamond$

The use of ISA and RISA can differentiate between possible class semantics. For an example, refer to Figure 5(b). If we declare that *collects* of *Only-painting collector* is restriction subclass of *collects* of *Art collector*, then the semantics of *Only-painting collector* will be as in *Option* 1 (i.e. an only-painting collector collects only paintings). On the other hand, if we declare that *collects* of *Painting collector* is just subclass (and not restriction subclass) of *collects* of *Art collector*, then the semantics of *Painting collector* will be as in *Option* 2.

The RISA relation defines a partial order over classes (up to equivalence), and has been studied extensively in [2, 3]. As we shall see in Sections 6 and 7, RISA is useful for relating inherited c-arrows to other arrows.

The following proposition expresses that any arrow inherited from an arrow class a' in a context m, is restriction subclass of a'.

Proposition 5.1. *Let a' be an arrow class such that $to(a')$ is instance of a context m, and let c be a subclass of $from(a')$. Then, $RISA(inh(c, a', m), a')$ holds.* $\diamond$

For example, in Figure 3(a), the inherited arrows $inh(c, a')$ and $inh(c, a', m)$ are restriction subclasses of a'.

The following proposition expresses that if a and a' are two arrow L_1-classes such that $\text{RISA}(a, a')$ holds, then the c-arrows inherited by an object o from a and a', in a context m, coincide.

Proposition 5.2. *Let a and a' be two arrow L_1-classes, and let m be a context such that $to(a)$ and $to(a')$ are instances of m. Let o be instance of $from(a)$. If $RISA(a, a')$ holds then the arrows $iInh(o, a, m)$ and $iInh(o, a', m)$ coincide.* $\diamond$

The following proposition describes the effect of subclass-inheritance on instance-inheritance.

Proposition 5.3. *Let a' be an arrow L_2-class such that context $to(a')$ is instance of a context m. Let c, c' be two instances of $from(a')$ such that c is subclass of c'. Then, the arrows $inh(c,\ iInh_2(c', a', m),\ m)$ and $iInh_2(c, a', m)$ coincide.* $\diamond$

6. Instance-Inheritance Relations

In this section, we introduce two new relations: the RES relation relates an instance-typing arrow with an arrow L_1-class, and the RES_2 relation relates an arrow L_1-class with an arrow L_2-class.

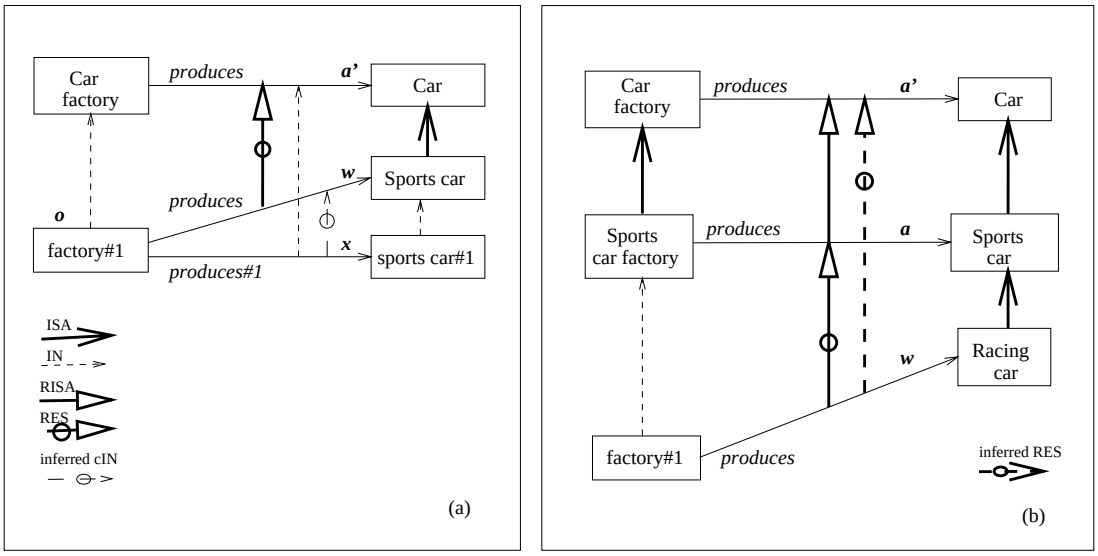

Figure 6. (a) The RES relation, (b) Interaction between the RES and RISA relations

We first present the RES relation that intuitively carries type information about an instance of a class. Let a' be an arrow L_1-class and let w be an instance-typing arrow with c-extension, say E. Roughly speaking, we say that w is *instance restriction* of a', denoted by $\text{RES}(w, a')$, if

E is the set of all instances of a' with $from$ object $from(w)$. For example, in Figure 6(a), let a' be the arrow *produces* from *Car-factory* to *Car*. Let w be the instance-typing arrow from *factory#1* to *Sports car* that refers collectively to the set of arrow tokens from *factory#1* to the sports cars it produces. If all cars produced by *factory#1* are sports cars then $\mathrm{RES}(w, a')$ holds. Thus, the relation $\mathrm{RES}(w, a')$ carries type information about *factory#1*.

Definition 6.1. RES relation
Let w be an instance-typing arrow, and let a' be an arrow L_1-class. We say that w is *instance-restriction* (or simply *restriction*) of a', denoted by $\mathrm{RES}(w, a')$, iff the following hold:
(i) $\mathrm{IN}(from(w), from(a'))$ and $\mathrm{ISA}(to(w), to(a'))$,
(ii) for any arrow x, if $\mathrm{cIN}(x, w)$ then $\mathrm{IN}(x, a')$, and
(iii) for any arrow x, if $\mathrm{IN}(x, a')$ and $from(x) = from(w)$ then $\mathrm{cIN}(x, w)$. $\diamond$

For example, in Figure 6(a), the relation $\mathrm{RES}(w, a')$ implies that $\mathrm{cIN}(x, w)$ holds.

Note that the definition of RES coincides with that of RISA if we replace cIN by IN in (ii) and (iii), IN by ISA in (i), and $from(x) = from(w)$ by $\mathrm{IN}(from(x), from(w))$ in (iii).

The following proposition shows the strong relation between the RES and RISA relations.

Proposition 6.1. *Let w be an instance-typing arrow, and let a, a' be two arrow L_1-classes. If RES(w, a) and RISA(a, a') hold then RES(w, a') holds.* $\diamond$

To illustrate Proposition 6.1, consider the example of Figure 6(b). Let w be the instance-typing arrow *produces* from *factory#1* to *Racing car*. The relation $\mathrm{RES}(w, a)$ expresses that all the sports cars produced by *factory#1* are racing cars. As $\mathrm{RES}(w, a)$ and $\mathrm{RISA}(a, a')$, it folllows that $\mathrm{RES}(w, a')$ holds. This expresses that all cars produced by *factory#1* are racing cars. Thus, the RISA relation can be used to derive additional type information about an instance.

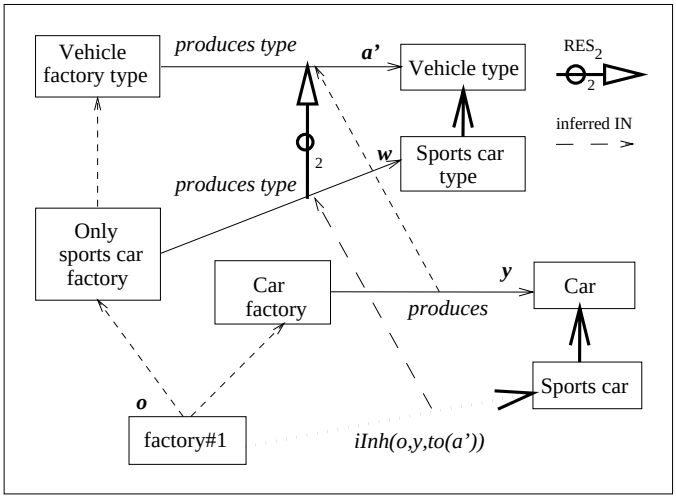

Figure 7. The RES_2 relation

We will now define the RES_2 relation that resembles the RES relation, but it holds at a higher abstraction level. Intuitively, the RES_2 relation enables retrieval of inherited c-arrows, in a context. Let a' be an arrow L_2-class such that $to(a')$ is a context, and let w be an arrow L_1-class with extension, say E. Roughly speaking the relation $\text{RES}_2(w, a')$ holds, if E is the set of c-arrows inherited by instances of $from(w)$ from instances of a', in the context of $to(a')$. Thus, retrieval of the c-arrows inherited by instances of $from(w)$ from instances of a', is enabled by simply querying for the instances of w.

Definition 6.2. RES_2 relation
Let w be an arrow L_1-class, and let a' be an arrow L_2-class such that $to(a')$ is a context. We say that w is a *contextual instance-restriction* (or simply *c-restriction*) of a', denoted by $\text{RES}_2(w, a')$, iff the following hold:
(i) $\text{IN}(from(w), from(a'))$ and $\text{ISA}(to(w), to(a'))$,
(ii) for any arrow x, $\text{IN}(x, w)$ holds iff
$\exists y$ such that $\text{IN}(y, a')$, $\text{IN}(from(x), from(w))$, and $x = iInh(from(x), y, to(a'))$, $\diamond$

For example, in Figure 7[2], let $\text{RES}_2(w, a')$ hold. Then, we can say that in the context of *Vehicle type*, every vehicle category produced by $factory\#1$ is instance of *Sports-car type*. In particular, as y is instance of a', and $factory\#1$ is instance of $from(w)$, it follows that the c-arrow inherited by $factory\#1$ from y in *Vehicle type*, is instance of w. Additionally, if x is an arrow of $factory\#1$ which is instance of w then there is an instance y of a' such that x is the c-arrow inherited by $factory\#1$ from y in the context $to(a')$.

The following proposition expresses that any c-arrow inherited by an object from an arrow L_1-class a', is restriction of a'. Additionally, any m-arrow inherited by a class from an arrow L_2-class a', is c-restriction of a'.

Proposition 6.2. *Let a' be an arrow class such that $to(a')$ is instance of a context m. Let o be instance of $from(a')$. Then, we have:*
(i) If a' is an arrow L_1-class then $RES(iInh(o, a', m), a')$.
(ii) If a' is an arrow L_2-class and $to(a')$ is a context then $RES_2(iInh_2(o, a', m), a')$. $\diamond$

For example, in Figure 4, the arrows $iInh(o, y_1, d'')$ and $iInh(o, y_2, d'')$ are restrictions of y_1, y_2, respectively. Additionally, the arrow $iInh_2(c', a'')$ is c-restriction of a''.

7. Instance-Inheritance Inference Rules

The fragment of the real world represented 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.

[2]In the already complicated figure, we do not show the relations $\text{IN}(from(y), from(a'))$ and $\text{IN}(to(y), to(a'))$.

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 $\text{IN}(o, c)$ holds, whereas the converse is not always true. Similarly, we denote by Isa, $Risa$, cIn, and Res, the subsets of the real world relations ISA, RISA, cIN, and RES, respectively, that are represented in the model.

In this section, we present a number of inference rules, called $iInh$ Rules, for deriving instance-inherited properties, as well as their relations to other properties based on declared relations. We also present an inference rule, called *meet* Rule, expressing that if an individual class c is subclass of two individual classes c' and c'' then c is subclass of the *meet* of c' and c''. Through the $iInh$ Rules and the *meet* Rule, the value domain of the instance-inherited properties is specialized. We present two complex examples demonstrating that these inference rules can give interesting schema derivations. The soundness of these inference rules with respect to the definitions given in the paper are given in Appendix B.

In the inference rules, we denote by I, A, X, IC, AC_1, and AC_2, the sets of individuals, arrows, contexts, individual classes, arrow L_1-classes, and arrow L_2-classes whose *to* object is a context, respectively. Additionally, we denote by S_i, for $i \geq 2$, the individual metaclass that refers collectively to all individual L_{i-1}-classes. Similarly, for uniformity reasons, the objects $iInh(o, a', m)$, $iD(o, a', m)$, and the relation Res are sometimes denoted by $iInh_1(o, a', m)$, $iD_1(o, a', m)$, and Res_1, respectively.

$iInh$ RULES

Rule 1: $\forall\, k \in \{1, 2\},\ a' \in AC_k,\ o \in I, m \in X,$

$$In(o, from(a')) \ \wedge\ In(to(a'), m) \ \Rightarrow \exists\, iInh_k(o, a', m) \in A \ \wedge\ \exists\, iD_k(o, a', m) \in IC \ \wedge$$
$$from(iInh_k(o, a', m)) = o \ \wedge\ to(iInh_k(o, a', m)) = iD_k(o, a', m)$$

Rule 2: $\forall\, k \in \{1, 2\},\ a' \in AC_k,\ o \in I,\ m \in X,$

$$In(o, from(a')) \ \wedge\ In(to(a'), m) \ \Rightarrow\ Res_k(iInh_k(o, a', m),\ a')$$

Rule 3: $\forall\, a_0, a, a' \in AC_1,\ o \in I,\ m \in X,$

$$Res(iInh(o, a_0, m), a) \ \wedge\ Risa(a, a') \ \Rightarrow\ Res(iInh(o, a_0, m),\ a')$$

$$\forall\, a_0, a, a' \in AC_1,\ o \in I,\ m \in X,$$
$$Res(iInh(o, a_0, m), a') \ \wedge\ Risa(a, a') \ \wedge\ In(o, from(a)) \ \wedge\ In(to(a), m)$$
$$\Rightarrow\ Res(iInh(o, a_0, m),\ a)$$

Rule 4: $\forall\, k \in \{1, 2\},\ a, a' \in AC_k,\ o \in I,\ m \in X,$

$$Res_k(iInh_k(o, a, m), a') \ \Rightarrow\ Isa(iD_k(o, a, m), to(a'))$$

Rule 5: $\forall\ a, a' \in AC_1,\ w \in A,\ o \in I,\ m \in X,$

$Res(iInh(o, a, m),\ a')\ \wedge\ Res(w, a')\ \wedge\ from(w) = o\ \wedge\ In(to(w), m)$

$\Rightarrow Isa(iD(o, a, m), to(w))$

$\forall\ a, a' \in AC_2,\ w \in AC_1,\ c \in IC,\ m \in X,$

$Res_2(iInh_2(c, a, m),\ a')\ \wedge\ Res_2(w, a')\ \wedge\ from(w) = c\ \wedge\ In(to(w), m)$

$\Rightarrow Isa(iD_2(c, a, m), to(w))$

Rule 6: $\forall\ a_0, a_1, a' \in AC_1,\ o \in I,\ m \in X,$

$Res(iInh(o, a_0, m), a')\ \wedge\ Res(iInh(o, a_1, m), a')\ \Rightarrow\ iInh(o, a_0, m) = iInh(o, a_1, m)$

$\forall\ a_0, a_1, a' \in AC_2,\ c \in IC,\ m \in X,$

$Res_2(iInh_2(c, a_0, m), a')\ \wedge\ Res_2(iInh_2(c, a_1, m), a')\ \Rightarrow\ iInh_2(c, a_0, m) = iInh_2(c, a_1, m)$

Rule 7: $\forall\ w, y \in AC_1,\ a' \in AC_2,$

$Res_2(w, a')\ \wedge\ In(y, a')\ \wedge\ In(o, from(y))\ \wedge\ In(o, from(w))$

$\Rightarrow\ In(iInh(o, y, to(a')), w)$

Rule 8: $\forall\ k \in \{1, 2\},\ a' \in AC_k,\ o \in I,\ i > k$

$In(o, from(a'))\ \wedge\ In(to(a'), S_i)\ \Rightarrow\ iInh_k(o, a') = iInh_k(o, a', S_i)$

---

meet RULE

Rule 1: $\forall\ c, c', c'' \in IC,\ \ Isa(c, c')\ \wedge\ Isa(c, c'')\ \Rightarrow\ Isa(c, meet(c', c''))$

In Figure 8, we give a complex example that illustrates the use of the above inference rules in deriving instance-inherited properties, and their relations to other properties.

The class *Only-car-factory type* refers to all types of factories that produce only cars. The class *Only-sports-car-factory* refers to all factories that produce only sport cars. The class *Convertible-sports-car-factory* refers to all factories that produce only sport cars including convertible-sports cars. The class *Very-fast-sports-car-factory* refers to all factories that produce only sport cars including very-fast-sports cars. The class *Racing-car-factory* refers to all factories that produce only sport cars, and the only very-fast-sports cars that they produce are racing cars.

Assume that the user has made the following declarations.

Declarations

1. The In and Isa declarations shown in the figure, as well as, the following:
   $In(a_1, a')$, $In(from(a_1), from(a'))$, $In(to(a_1), to(w_2))$,

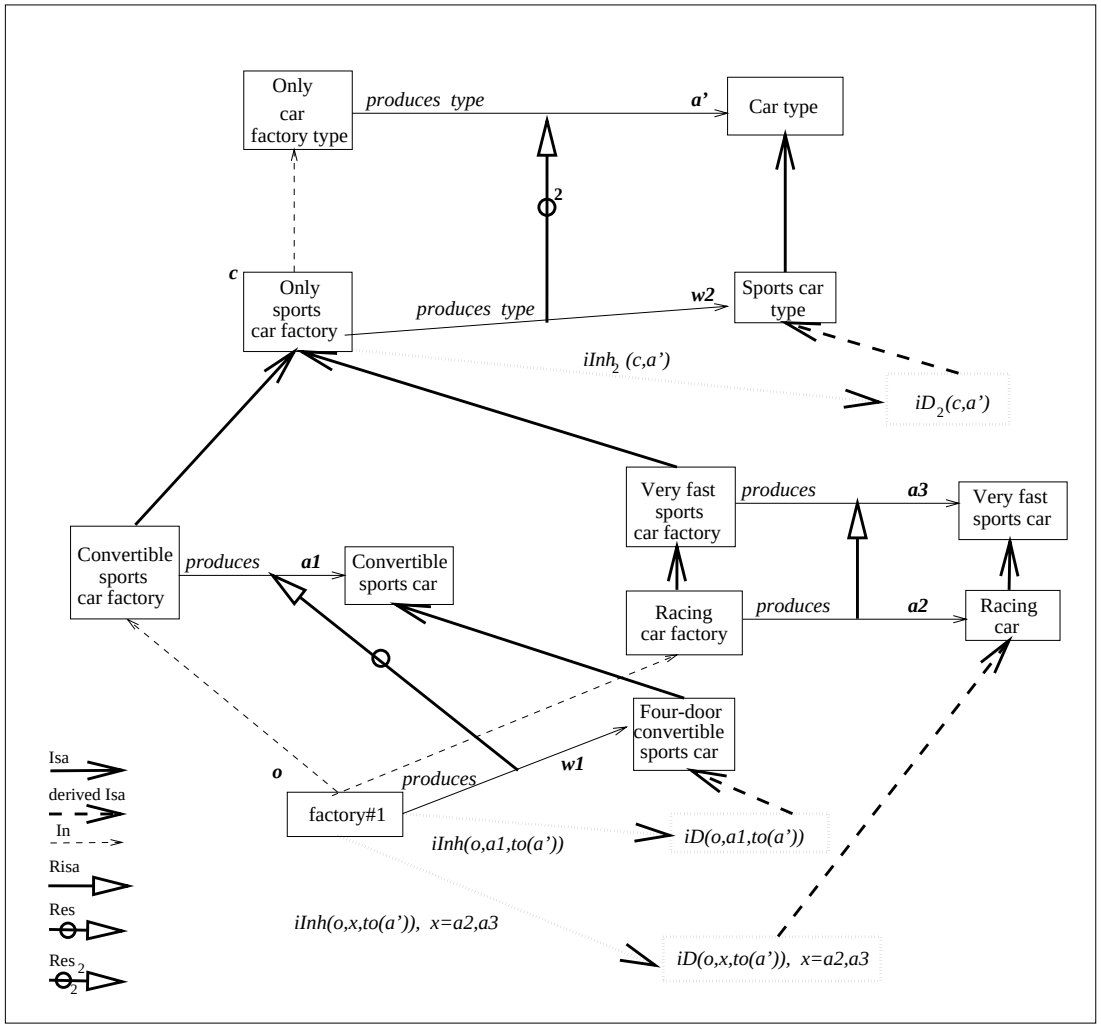

Figure 8. A complex example demonstrating the inference rules

$In(a_2, a')$, $In(from(a_2), from(a'))$, $In(to(a_2), to(w_2))$,
and $In(a_3, a')$, $In(from(a_3), from(a'))$, $In(to(a_3), to(w_2))$.

2. The individual L_2-class $to(a')$ is a context.
3. $Res_2(w_2, a')$, expressing that in the context of *Car type*, every car category produced by an only-sports-car-factory is a sports-car type (i.e. instance of *Sports-car type*). Additionally, it expresses that w_2 refers collectively to the c-arrows inherited by instances of c from instances of a', in the context of *Car type*.
4. $Risa(a_2, a_3)$, expressing that every very-fast-sports car produced by a racing-car-factory is a racing car.
5. $Res(w_1, a_1)$, expressing that every convertible-sports car produced by factory#1 has four doors.

Using the $iInh$ Rules and the above declarations, we can now obtain the following derived relations.

Derived Relations

1. The arrow $iInh(o, a_1, to(a'))$ is defined and it holds that $Isa(iD(o, a_1, to(a')), to(w_1))$. This expresses that the category of convertible-sports cars produced by $factory\#1$ is subclass of *Four-door-convertible-sports car*. In other words, all the convertible-sports cars produced by $factory\#1$ have four doors.
2. The arrows $iInh(o, a_2, to(a'))$ and $iInh(o, a_3, to(a'))$ coincide. Additionally, it holds that $Isa(iD(o, x, to(a')), to(a_2))$, for $x = a_2, a_3$. These derived relations express that the categories of very-fast-sports cars and racing cars produced by $factory\#1$ are the same in the context *Car type*. Additionally, this common category is subclass of *Racing car*. Therefore, all the very fast sports cars produced by $factory\#1$ are racing cars.
3. The arrow $iInh_2(c, a')$ is defined and it holds that $Isa(iD_2(c, a'), to(w_2))$. This derived relation expresses that the class referring collectively to the car categories produced by an only-sports-car factory is subclass of *Sports-car type*. In other words, each car type produced by an only-sports-car factory is a sports-car type.
4. The arrows $iInh(o, a_1, to(a'))$, $iInh(o, a_2, to(a'))$, and $iInh(o, a_3, to(a'))$ are instances[3]. of the arrow $iInh_2(c, a')$ and the arrow w_2.

We will also now demonstrate the application of the $iInh$ Rules to the Example of Figure 1. We will show that using the $iInh$ Rules, we can derive useful type information about *factory#2*. Assume that the arrows *produces* of *Convertible-car factory*, and *produces* of *Sports-car factory* are restriction subclasses of the arrow *produces* of *Car factory*. That is, the relations $Risa(a_1, a')$ and $Risa(a_2, a')$ hold. These relations express that every car produced by a convertible-car factory is a convertible-car, and that every car produced by a sports-car factory is a sports-car. Additionally, assume that the instance-typing arrow *produces* of *factory#2* is restriction of the arrow *produces* of *Car factory*. That is, the relation $Res(w, a')$ holds. This relation expresses that every car produced by *factory#2* is a racing car. Assume that all individual classes whose name finishes in "car" are instances of *Vehicle type* (denoted by m).

From $iInh$ Rule 2, it is derived that $iInh(o, a', m)$[4] is restriction of a'. Additionally, from the declared relations $Risa(a_1, a')$ and $Risa(a_2, a')$, and from $iInh$ Rule 3, it is derived that $iInh(o, a', m)$ is restriction of a_1 and a_2. From these derived Res relations and $iInh$ Rule 4, it is derived that $iD(o, a', m)$ is subclass of *Convertible-car* and *Sports-car*. dditionally, from the declared relation $Res(w, a')$ and $iInh$ Rule 5, it is derived that $iD(o, a', m)$ is subclass of *Racing-car*. Assume that *Convertible-racing car* is the *meet* of *Convertible-car* and *Sports-car*. From the derived Isa relations and *meet* Rule 1, it is derived that $iD(o, a', m)$ is subclass of *Convertible-racing car*. This expresses that every car produced by *factory#2* is a convertible-racing car.

[3]These derived In relations are not shown in Figure 8, in order not to further complicate the figure.

[4]This inherited c-arrow is not shown in Figure 1.

8. Related Work

The form of instance-inheritance supported by our data model is *monotonic*, in the sense that any new declaration can only specialize the value of an instance-inherited property, but cannot "override" that property. Monotonic instance-inheritance is supported by terminological languages [7, 5, 24, 21, 4] and by several deductive object-oriented models, such as DOT [27, 26], and QUIXOTE [28, 29]. In contrast, several systems, such as [6, 14, 20, 1, 13, 17], support *non-monotonic* instance-inheritance, where the value of an instance-inherited property can be overridden over subclasses, or over particular instances. In this section, we briefly compare our approach to instance-inheritance to those mentioned above. In our comparison, we use the example of Figure 1, where we assume that (i) the arrows *produces* of *Convertible-car factory* and *Sports-car factory* are related to the arrow *produces* of *Car factory* by *Risa*, and (ii) the instance-typing arrow *produces* of *factory#2* is related to the arrow *produces* of *Car factory* by *Res*.

8.1. Terminological languages

Models based on terminological languages [7, 5, 24, 21, 4] support taxonomic representation of *concepts* and *objects*, that correspond respectively to individual L_1-classes and individual tokens, in our model. Both, concepts and objects, are described in terms of other concepts and necessary and sufficient conditions on their properties, through a set of operators. An object is instance of all the concepts it satisfies. Both, concepts and objects, are put into a hierarchy where a concept is below the concepts it specializes, and an object is below the concepts it satisfies. Concepts and objects inherit properties and property value constraints from the concepts above in the hierarchy. Local and inherited property value constraints are combined to determine the value of the property.

For example, in Figure 1, the concept *Car-factory* is described as *(ALL produces Car)*, meaning that an instance of *Car-factory* is an object whose *all* properties *produces*[5] have values in *Car*.

The concept *Sports-car-factory* is described as (AND *Car-factory* (ALL *produces Sports-car*)), meaning that an instance of *Sports-car-factory* is an instance of *Car-factory* whose *all* properties *produces* have values in *Sports-car*. Similarly, the concept *Convertible-car factory* is described as (AND *Car-factory* (ALL *produces Convertible-car*)). The concept *Convertible-sports-car-factory* is described as (AND *Convertible-car-factory Sports-car-factory*). The object *factory#1* is described as instance of *Convertible-sports-car-factory*, and the object *factory#2* is described as instance of the concept (AND *Convertible-sports-car-factory* (ALL *produces Racing-car*)).

[5] In our data model, the expression "all properties *produces*" means "all properties that are instances of the property *produces*". In terminological models, the fact that a property of an object is instance of a property of a concept is *implied* by the use of the same name for both properties. We will see below that this is a limitation, as, in contrast to our model, it does not allow for a property of an object to be instance of two dinstinct properties with different semantics.

Terminological models, similarly to our model, will derive that all cars produced by *factory#1* are convertible-sports cars. In other words, they will derive that *factory#1* is also instance of the concept (ALL *produces Convertible-sports-cars*). Additionally, they will derive that *factory#2* is also instance of the concept (ALL *produces Convertible-racing-cars*), meaning that all cars produced by *factory#2* are convertible-racing cars. Thus, terminological models will give the same answer as our model to queries such as "what is the category of cars produced by *factory#1*.

However, compared to our model, terminological models have certain limitations. For example, if two concepts have each a property with the same name, then it is implied that the two properties have the same semantics[6]. However, this may not give the desirable results. For example, consider the arrow *produces* from *Car-factory* to *Car*, and the arrow *produces* from *Boat-factory* to *Boat*. Additionally, assume that *Car-boat-factory* is the meet of *Car-factory* and *Boat-factory*. In our model, the class *Car-boat-factory* inherits two different arrows: one from the arrow *produces* of *Car-factory*, and another from the arrow *produces* of *Boat-factory*. This expresses that a car-boat factory can produce both cars and boats (and not that a car-boat factory produces only objects that are both car and boat). In contrast, terminological models will derive that *Car-boat-factory* is specialization of *(ALL produces (AND Car Boat))*, meaning that a car-boat factory produces only objects that are both car and boat (clearly, not the desirable result). Additionally, consider a car-boat factory o with a property *produces* whose value is o_1. Terminological models can only express the case where o_1 is both car and boat. This case can also be expressed in our model by declaring that a_1 is instance of both arrows: the arrow *produces* of *Car-factory* and the arrow *produces* of *Boat-factory*. However, the case where o_1 *is not* a boat cannot be expressed in terminological models, whereas it can be expressed in our model (by declaring that a_1 is instance of the arrow *produces* of *Car-factory*, only).

In terminological models, the user can query about the most specific concept satisfied by the values of a property of a particular object. However, he cannot query about this most specific concept, in the context of a metaclass. This is because, terminological models do not support metaclasses, and therefore they cannot support concepts, such as context and inherited m-arrow.

Additionally, in terminological models, there is no way to refer collectively to the associations between objects of a concept and the most specific concept satisfied by the values of several properties of these objects. In contrast, this can be done in our model through the Res_2 relation.

8.2. DOT & QUIXOTE

The knowledge representation models DOT [27, 26] and QUIXOTE [29] describe property values using the Isa relation. In these models, the In relation is not distinguished from the Isa relation (for this reason, instead of the term *subclass*, we will use the term *specialization* when we refer to DOT and QUIXOTE). For example, in Figure 1, the In relation should be replaced by Isa. The value of a property with label p of an object c is denoted by $c.p$. Property inheritance is

[6] In our model, this means that they are related through RISA to a common, more generic property (e.g., the properties *produces* of *Convertible car* and *Sports car* of Figure 1 have the same semantics).

supported through the following inference rules[7]:
(i) if $Isa(c, c')$ then $Isa(c.p,\ c'.p)$, and
(ii) if $Isa(c.p,\ d)$ and $Isa(c.p,\ d')$ then $Isa(c.p,\ meet(\{d, d'\})$.
Using these inference rules, it will be derived that *factory#1.produces* is specialization of *Convertible-sports car*, and *factory#2.produces* is specialization of *Convertible-racing car*. However, as the *In* and *Isa* relations are not distinguished, it is difficult to find out if *Convertible-racing car* corresponds to a particular car produced by *factory#2*, or to categorical information. In addition, the limitations we mentioned for the terminological models also apply to DOT and QUIXOTE.

8.3. Shared-value properties and default properties

In the literature, inheritance from a class to its instances has been considered in connection with *shared-value properties* [15, 23] and *default properties* [6, 14, 20, 1, 13, 17][8]. A shared-value property is a property with the same value for all instances of a class. Such a property is associated with the class, and not with particular instances of the class. Then, particular instances inherit the property from the class. For example, *has-material* is a shared-value property of the class *Copper pot* with value *Copper*. This property is inherited by *every* copper pot.

A default property, on the other hand, is a shared-value property of a class that can be "overridden" over subclasses, or over particular instances of the class. For example, we can say that the property *has-material* of *Golden bracelet* with value *Gold* is a default property. Indeed, the value *Gold* is overridden over certain subclasses of *Bracelet*, such as *14K-Golden bracelet*, where the value *Gold* becomes *14K-Gold*. In our data model, overriding takes place only through value specialization. However, in general, overriding does not imply value specialization. For example, assume that the property classes in Figure 1 are default properties. Then, the value of the property *produces* inherited by *factory#2* is *Racing car* (and not *Convertible-racing car*, as in our data model). Additionally, in the case of multiple specialization and multiple instantiation, inheritance of default properties may cause ambiguities. For example, the value of the property *produces* inherited by *factory#1* is ambiguous, as neither of the classes *Convertible-car factory* and *Sports-car factory* is more specific than the other.

Although, in this example, default property inheritance does not give the desirable result, it is nevertheless useful in certain applications. For example, consider the class *Patient* and its subclass *Alcoholic*, and assume that *Patient* has a property *has-doctor* with value *Physician*, and that *Alcoholic* has a property *has-doctor* with value *Psychologist*. Note that *Psychologist* is not subclass of *Physician*, as in general a psychologist is not a physician. We woule like the property *has-doctor* of *Alcoholic* to "override" the property *has-doctor* of *Patient*. In other words, we

[7] In fact, DOT supports only inference rule (i).

[8] In these models, default rules are attached to a class, and are used to compute property values for each instance of the class. The default rules attached to a class are overridden by rules attached to subclasses or instances of the class. A default property of a class can be seen as a special case of default rule, where the computed property value is the same for all instances of the class.

would like the value of the property inherited by an instance of *Alcoholic* from the property *has-doctor* to be *Psychologist*. This can be achieved by default property inheritance, while it cannot be achieved in our data model (as *Psychologist* is not subclass of *Physician*).

9. Conclusions

In this paper, we have introduced the concept of instance-typing arrows with the objective to model type information about particular class instances. Type properties of classes carry type information about all instances of a class. In contrast, instance-typing properties carry type information about particular instances.

We have presented a framework which provides support for inheritance of arrows from a class to an instance, using type information about the class, as well as type information about the instance. This kind of inheritance provides us with the most specific type information about the instance. This information is expressed as the value of derived arrows that we called *inherited c-arrows*.

We have also introduced the concept of *inherited m-arrows*. An inherited m-arrow refers collectively to a set of inherited c-arrows, thus allowing their retrieval in the same way as for "usual" arrows. In addition, the value of an inherited m-arrow provides summary information about the values of the inherited c-arrows to which it refers.

Both inherited c-arrows and inherited m-arrows, as well as their relations to other arrows, are derived through a set of inference rules.

A novel characteristic of our approach is that inheritance takes place in a context. The need to introduce a context comes from the fact that we would like the value of the inherited arrow to be instance of a metaclass of interest. We are currently investigating a broader notion of context in which the metaclass can be an arrow metaclass (and not just an individual metaclass). Another line of research would be to generalize the concept of categorical instance-inheritance from arrow L_1-classes to arrow meta-classes.

APPENDIX A: Proofs of Propositions

Here, we give the proofs of all propositions.

Proposition 5.1 Let a' be an arrow class such that $to(a')$ is instance of a context m and let c be a subclass of $from(a')$. Then, RISA$(inh(c, a', m), a')$ holds.

Proof:
Let x be an arrow such that IN$(x, inh(c, a', m))$. Then, IN(x, a'). Therefore, it follows that EXT$(inh(c, a', m)) \subseteq$ EXT(a'), and ISA$(D(c, a'), to(a'))$. As IN$(to(a'), m)$, it follows that ISA$(D(c, a', m), to(a'))$. As $from(inh(c, a', m)) = c$, $to(inh(c, a', m)) = D(c, a', m)$, it now follows that ISA$(inh(c, a', m), a')$.

Additionally, let x be an arrow such that IN(x, a') and IN$(from(x), c)$ then it holds IN$(x, inh(c, a', m))$. It now follows that RISA$(inh(c, a', m), a')$. $\diamond$ $\square$

Proposition 5.2 Let a and a' be two arrow L_1-classes, and let m be a context such that $to(a)$ and $to(a')$ are instances of m. Let o be instance of $from(a)$. If RISA(a, a') holds then the arrows $iInh(o, a, m)$ and $iInh(o, a', m)$ coincide.

Proof:
To show that $iInh(o, a, m)$ and $iInh(o, a', m)$ coincide, it is enough to show that they have the same c-extensions and the same *to* objects. Let x be an arrow such that cIN$(x, iInh(o, a, m))$. Then, from Def. 4.2, it follows that IN(x, a). As $Isa(a, a')$, it follows that IN(x, a'). This implies that cIN$(x, iInh(o, a', m))$.

Conversely, let x be an arrow such that cIN$(x, iInh(o, a', m))$. Then, it holds IN(x, a'). As $from(x) = o$, it follows that $from(x)$ is instance of $from(a)$. As $Risa(a, a')$, it follows that IN(x, a). It now follows that cIN$(x, iInh(o, a, m))$. Therefore, $iInh(o, a, m)$ and $iInh(o, a', m)$ have the same c-extension. From this, it follows that the classes $iD(o, a)$ and $iD(o, a')$ coincide. Therefore, the classes $iD(o, a, m)$ and $iD(o, a', m)$ coincide. We have thus proved that $iInh(o, a, m)$ and $iInh(o, a', m)$ coincide. $\diamond$ $\square$

Proposition 5.3 Let a' be an arrow L_2-class such that context $to(a')$ is instance of a context m. Let c, c' be two instances of $from(a')$ such that c is subclass of c'. Then, the arrows $inh(c,\ iInh_2(c', a', m),\ m)$ and $iInh_2(c, a', m)$ coincide. $\diamond$

Proof:
Obviously, the arrows $inh(c,\ iInh_2(c', a', m),\ m)$ and $iInh_2(c, a', m)$ are specialized. Let x be an instance of $inh(c,\ iInh_2(c', a', m),\ m)$. Then, x is also instance of $iInh(c', a', m)$. Thus, there is y such that IN(y, a') and $x = iInh(from(x), y, to(a'))$. As $from(x)$ is instance of c, it follows that x is also instance of $iInh_2(c, a', m)$.

Let x be an instance of $iInh_2(c, a', m)$. Then, $from(x)$ is instance of c, and thus, $from(x)$ is instance of c'. Additionally, there is y such that IN(y, a') and $x = iInh(from(x), y, to(a'))$. Thus, x is also an instance of $iInh_2(c', a', m)$. As $from(x)$ is instance of c, it follows that x is also instance of $inh(c,\ iInh_2(c', a', m),\ m)$.

Therefore, the arrows $inh(c,\ iInh_2(c', a', m),\ m)$ and $iInh_2(c, a', m)$ have the same extension. Obviously, $D(c,\ iInh_2(c', a', m))$ and $iD_2(c, a')$ coincide. However, $D(c,\ iInh_2(c', a', m),\ m)$ is the cover of $D(c,\ iInh_2(c', a', m))$ in m and $iD_2(c, a', m)$ is the cover of $iD_2(c, a')$ in m. Therefore, the classes $D(c,\ iInh_2(c', a', m),\ m)$ and $iD_2(c, a', m)$ coincide. $\diamond$ $\square$

Proposition 6.1 Let w be an instance-typing arrow, and let a, a' be two arrow L_1-classes. If RES(w, a) and RISA(a, a') hold then RES(w, a') holds.

Proof:
Let x be an arrow such that cIN(x, w). Then, from Def. 6.1, it follows that IN(x, a). As ISA(a, a'), it follows that IN(x, a').

Let x be an arrow such that IN(x, a') and $from(x) = from(w)$. As RISA(a, a'), it follows that IN(x, a). Fom Def. 6.1, it now follows that cIN(x, w). Therefore, it holds RES(w, a'). $\diamond$ $\square$

Proposition 6.2 Let a' be an arrow class such that $to(a')$ is instance of a context m. Let o be instance of $from(a')$. Then, we have:
(i) If a' is an arrow L_1-class then RES$(iInh(o, a', m), a')$ holds.
(ii) If a' is an arrow L_2-class such that $to(a')$ is a context then RES$_2(iInh_2(o, a', m), a')$ holds.

Proof:
(i) Let x be an arrow such that cIN$(x, iInh(o, a', m))$. It follows from Def. 4.2 that IN(x, a'). Let x be an arrow such that IN(x, a') and $from(x) = o$. It follows from Def. 4.2 that cIN$(x, iInh(o, a', m))$.

Therefore, it holds RES$(iInh(o, a', m),\ a')$.
(ii) (Condition (i) of Def. 6.2) From assumptions, it holds that o is instance of $from(a')$. Additionally, it holds that the class $iD_2(o, a', m)$ is subclass of $to(a')$.
(Condition (ii) of Def. 6.2) Let x be an arrow such that IN$(x, iInh_2(o, a', m))$. It follows from Def. 4.4 that there is an arrow y such that IN(y, a') and $x = iInh(from(x), y, to(a'))$.

Let x be an arrow such that IN$(from(x), o)$. Additionally, let y be an arrow such that IN(y, a'), and $x = iInh(from(x), y, to(a'))$. It follows from Def. 4.4 that IN$(x, iInh_2(o, a', m))$.

Therefore, it holds RES$_2(iInh(o, a', m),\ a')$. $\diamond$ $\square$

APPENDIX B: Soundness of Inference Rules

Here, we prove the soundness of the inference rules presented in section 7.

iInh Rule 1

Directly from Def. 4.2 and Def. 4.4. $\diamond$

iInh Rule 2

Directly from Proposition 6.2. $\diamond$

iInh Rule 3

First rule.

Directly from Proposition 6.1.

Second rule.

Let x be an arrow such that cIN$(x, iInh(o, a_0, m))$. As $Res(iInh(o, a_0, m), a')$, it follows from Def. 6.1 that IN(x, a'). As $from(x) = o$ and $In(o, from(a))$, it follows that IN$(from(x), from(a))$. As $Risa(a, a')$, it follows that IN(x, a).

Let x be an arrow such that IN(x, a) and $Isa(from(x), o)$. As $Isa(a, a')$, it follows that IN(x, a'). As $Res(Inh(o, a_0, m), a')$, it follows that cIN$(x, iInh(o, a_0, m))$.

We have shown that if x is an arrow such that cIN$(x, iInh(o, a_0, m))$ then IN(x, a). Thus, $Isa(iD(o,$ $a_0)$, $to(a))$. As $In(to(a), m)$, it follows that $Isa(iD(o,\ a_0, m), to(a))$.

Therefore, it holds $Res(iInh(o, a_0, m),\ a)$. $\diamond$

iInh Rule 4

Directly from Def. 6.1 and Def. 6.2. $\diamond$

iInh Rule 5

First rule.

Let x be an arrow such that cIN$(x, iInh(o, a_0, m))$. As $Res(Inh(o, a_0, m), a')$, it follows from Def. 6.1 that IN(x, a'). As $from(x) = o$, it follows that $from(x) = from(w)$. As $Res(w, a')$, it follows that cIN(x, w). From this, it follows that IN$(to(x), to(w))$. Therefore, it follows that ISA$(iD(o, a_0), to(w))$. As $In(to(w), m)$, it follows that $Isa(iD(o, a_0, m), to(w))$.

Second rule.

Let x be an arrow such that IN$(x, iInh_2(c, a_0, m))$. As $Res_2(Inh_2(c, a_0, m), a')$, it follows from Def. 6.2 that there is an arrow y such that IN(y, a') and $x = iInh(from(x), y, to(a'))$. As $from(w) = c$, it follows that IN$(from(x), from(w))$. As $Res_2(w, a')$, it follows from Def. 6.2 that IN(x, w). From this, it follows that ISA$(iD_2(c, a_0), to(w))$. As $In(to(w), m)$, it follows that $Isa(iD(o, a_0, m), to(w))$. $\diamond$

iInh Rule 6

First rule.

Let x be an arrow such that cIN$(x, iInh(o, a_0, m))$. As $Res(Inh(o, a_0, m), a')$, it follows from Def. 6.1 that IN(x, a'). As $from(x) = o$ and $Res(iInh(o, a_1, m), a')$, it follows from Def. 6.1 that $cIn(x, iInh(o, a_1, m))$. Similarly, if x is an arrow such that cIN$(x, iInh(o, a_0, m))$ then cIN$(x, iInh(o, a_1, m))$. Thus, $iInh(o, a_0, m)$ and $iInh(o, a_1, m)$ have the same c-extension. From this, it is derived that $iD(o, a_0, m)$ and $iD(o, a_1, m)$ coincide. Therefore, $iInh(o, a_0, m)$ and $iInh(o, a_1, m)$ coincide.

Second rule.

Let x be an arrow such that IN$(x, iInh_2(c, a_0, m))$. As $Res_2(iInh_2(c, a_0, m), a')$, it follows from Def. 6.2

that there is an arrow y such that $\mathrm{IN}(y, a')$ and $x = iInh(from(x), y, to(a'))$. As $\mathrm{IN}(from(x), c)$ and $Res_2(iInh_2(c, a_1, m), a')$, it follows from Def. 6.2 that $\mathrm{IN}(x, iInh_2(c, a_1, m))$. Similarly, if x is an arrow such that $\mathrm{IN}(x, iInh_2(c, a_1, m))$ then $\mathrm{IN}(x, iInh_2(c, a_0, m))$. Thus, $iInh_2(c, a_0, m)$ and $iInh_2(c, a_1, m)$ have the same extension. Obviously, $iD_2(c, a_0) = iD_2(c, a_1)$. However, $iD_2(c, a_0, m)$ is the cover of $iD_2(c, a_0)$ in m, and $iD_2(c, a_1, m)$ is the cover of $iD_2(c, a_1)$ in m. Therefore, $iD_2(c, a_0, m)$ and $iD_2(c, a_1, m)$ coincide. Thus, the arrows $iInh_2(c, a_0, m)$ and $iInh_2(c, a_1, m)$ coincide. $\diamond$

iInh Rule 7
Directly from Definition 6.2. $\diamond$

iInh Rule 8
Case 1: $k = 1$.
Obviously, the arrows $iInh(o, a')$ and $iInh(o, a', S_i)$ are defined and have the same extensions. As $iD(o, a')$ is instance of S_i, it follows that $iD(o, a') = iD(o, a', S_i)$. Thus, $iInh(o, a') = iInh(o, a', S_i)$. $\diamond$
Case 2: $k = 2$.
Obviously, the arrows $iInh_2(o, a')$ and $iInh_2(o, a', S_i)$ are defined and have the same extensions. As $to(a')$ is instance of S_i, it follows that $iD_2(o, a')$ is instance of S_i. It now easily follows that $iD_2(o, a') = iD_2(o, a', S_i)$. Thus, $iInh_2(o, a') = iInh_2(o, a', S_i)$. $\diamond$

APPENDIX C: Proofs of the Derived Relations of Section 7

Here, we give the proofs of the derived relations concerning the example of Figure 8.

Derived Relations

1. The arrow $iInh(o, a_1, to(a'))$ is defined and it holds that $Isa(iD(o, a_1, to(a')), to(w_1))$.

   Proof: From the facts $In(o, from(a_1))$, and $In(to(a_1), to(a'))$, it is derived through $iInh$ Rule 1 that the arrow $iInh(o, a_1, to(a'))$ is defined. Additionally, it is derived through $iInh$ Rule 2 that $Res(iInh(o, a_1, to(a')), a_1)$. From this and the facts $Res(w_1, a_1)$, $from(w_1) = o$, and $In(to(a_1), to(a'))$, it is derived through $iInh$ Rule 5 that $Isa(iD(o, a_1, to(a')), to(w_1))$. $\diamond$

2. The arrows $iInh(o, a_2, to(a'))$ and $iInh(o, a_3, to(a'))$ coincide. Additionally, it holds that $Isa(iD(o, x, to(a')), to(a_2))$, for $x = a_2, a_3$.

   Proof: From the facts $In(o, from(a_3))$, and $In(to(a_3), to(a'))$, it is derived through $iInh$ Rule 1 that the arrow $iInh(o, a_3, to(a'))$ is defined. Additionally, it is derived through $iInh$ Rule 2 that $Res(iInh(o, a_3, to(a')), a_3)$. From this, and the facts $Risa(a_2, a_3)$ and $In(o, from(a_2))$, it is derived through $iInh$ Rule 3 that $Res(iInh(o, a_3, to(a')), a_2)$.

   From the facts $In(o, from(a_2))$ and $In(to(a_2), to(a'))$, it is derived through $iInh$ Rule 1 that the arrow $iInh(o, a_2, to(a'))$ is defined. Additionally, it is derived through $iInh$ Rule 2 that $Res(iInh(o, a_2, to(a')), a_2)$. As $Risa(a_2, a_3)$, it is derived from $iInh$ Rule 3 that $Res(iInh(o, a_2, to(a')), a_3)$. Similarly, it is derived that $Res(iInh(o, a_3, to(a')), a_3)$. Using the facts $Res(iInh(o, x, to(a')), a_3)$, for $x = a_2, a_3$, it is derived through $iInh$ Rule 6 that the arrows $iInh(o, x, to(a'))$, for $x = a_2, a_3$, coincide. Additionally, from the fact $Res(iInh(o, a_2, to(a')), a_2)$, it is derived through $iInh$ Rule 4 that the *to* object of these arrows is subclass of $to(a_2)$. $\diamond$

3. The arrow $iInh_2(c, a')$ is defined, and it holds that $Isa(iD_2(c, a'), to(w_2))$.

   Proof: Let S_4 be the metaclass that refers collectively to all individual L_3-classes. From the facts $In(c, from(a'))$, and $In(to(a'), S_4)$, it is derived through $iInh$ Rule 1 that the arrow $iInh_2(c, a', S_4)$, is defined. It is derived through $iInh$ Rule 2 that $Res_2(iInh_2(c, a', S_4), a')$.

   From the facts that $Res_2(w_2, a')$, $from(w_2) = c$, and $In(to(w_2), S_4)$, it is derived through $iInh$ Rule 5 that the class $iD_2(c, a', S_4)$, is subclass of $to(w_2)$. It is derived from $iInh$ Rule 7 that the arrows $iInh_2(c, a')$ and $iInh_2(c, a', S_4)$, coincide. Therefore, the class $iD_2(c, a')$ is subclass of $to(w_2)$. $\diamond$

4. The arrows $iInh(o, a_1, to(a'))$, $iInh(o, a_2, to(a'))$, and $iInh(o, a_3, to(a'))$ are instances of the arrow $iInh_2(c, a')$ and w_2.

   Proof: It is derived through $iInh$ Rule 2 that $Res_2(iInh_2(c, a', S_4), a')$ holds. As a_1, a_2, and a_3 are instances of a' and o is instance of the classes c, $from(a_1)$, $from(a_2)$, and $from(a_3)$, it is derived from $iInh$ Rule 7 that the arrows $iInh(o, a_1, to(a'))$, $iInh(o, a_2, to(a'))$, and $iInh(o, a_3, to(a'))$ are instances of the arrow $iInh_2(c, a', S_4)$. It is derived from $iInh$ Rule 8 that the arrows $iInh_2(c, a')$ and $iInh_2(c, a', S_4)$ coincide. Therefore, the arrows $iInh(o, a_1, to(a'))$, $iInh(o, a_2, to(a'))$, and $iInh(o, a_3, to(a'))$ are instances of the arrow $iInh_2(c, a')$.

   It has now been declared that $Res_2(w, a')$ holds. As a_1, a_2, and a_3 are instances of a' and o is instance of the classes c, $from(a_1)$, $from(a_2)$, and $from(a_3)$, it is derived from $iInh$ Rule 7 that the arrows $iInh(o, a_1, to(a'))$, $iInh(o, a_2, to(a'))$, and $iInh(o, a_3, to(a'))$ are also instances of the arrow w_2. $\diamond$

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] A. Analyti, N. Spyratos, P. Constantopoulos, On the Semantics of a Semantic Network, Fundamenta Informaticae, 36, 109-144 (1998).

[3] A. Analyti, P. Constantopoulos, N. Spyratos, Restriction by Specialization and Schema Derivations, *Information Systems*, 23(1), 1-38 (1998).

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

[5] 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).

[6] 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).

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

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

[9] 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/index.html

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

[11] 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).

[12] 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).

[13] G. Dobbie, R. W. Topor: A Model for Sets and Multiple Inheritance in Deductive Object-Oriented Systems, *Third International Conference on Deductive and Object-Oriented Databases (DOOD'93)*, 473-488 (1993).

[14] 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).

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

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

[17] 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).

[18] 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).

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

[20] F.G. McCabe, Logic and Objects, Prentice Hall (1992).

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

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

[23] A.H. Ngu, L. Wong, S. Widjojo, On Canonical and Non-canonical Classifications, *Second International Conference on Deductive and Object-Oriented Databases (DOOD'91)*, 371-390 (1991).

[24] 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).

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

[26] 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).

[27] 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).

[28] H. Yasukawa, H. Tsuda, K. Yokota, Objects, Properties, and Modules in QUIXOTE. *Proceedings of the International Conference on Fifth Generation Computer Systems (FGCS'92)*, 257-268 (1992).

[29] 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).