Fundamenta Informaticae 33 (1998) 1–34
IOS Press

On the Semantics of a Semantic Network

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

We elaborate on the *semantics* of an enhanced object-oriented semantic network, where multiple instantiation, multiple specialization, and meta-classes are supported for both kinds of objects: entities and properties. By semantics of a semantic network, we mean the information (both *explicit* and *derived*) that the semantic network carries. Several data models use semantic networks to organize information. However, many of these models do not have a formalism defining what the semantics of the semantic network is.

In our data model, in addition to the *Isa* relation, we consider 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 data model is that it supports the interaction between *Isa* and *Risa* relations. The combination of *Isa* and *Risa* provides a powerful conceptual modeling mechanism.

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

The user declares objects and relations between objects through a *program*. Reasoning is done through a number of (built-in) *inference rules* that allow for derivations both at instance and schema level. Through the inference rules, new objects and new relations between objects are derived. In our data model, inherited properties are considered to be derived objects. In addition to the inference rules, a number of (built-in) *system constraints* exist for checking the validity of a program.

Keywords: semantic network, semantics, inheritance, inference rules, constraints, conceptual modeling.

1. Introduction

Structure can carry useful and expressive information that can be deduced with high efficiency. This motivated the development of semantic networks [44, 20, 36] and the design of several useful structural mechanisms. In a semantic network, real world objects are represented by means of nodes and links. Here, by *real world objects* we mean entities, properties, or relationships that make up the user's perception of the real world.

Nodes and links are used to build semantically rich structures. These structures not only represent knowledge by themselves but are also used for deriving new information or for checking the validity of the existing information. We call the information (both explicit and derived) that a semantic network carries, the *semantics* of the network.

Several data models use semantic networks to organize information [12, 21, 39, 43, 7, 23, 34, 30, 19, 22, 9] and their usefulness to conceptual modeling is unquestionable. However, many of these models do not provide a formalism defining what the semantics of the semantic network is. This can lead to inconsistencies as (procedurally) derived information cannot be always validated against the (declaratively) defined semantics. Additionally, derivations and reasoning are limited, as they are procedurally determined.

In this work, we elaborate on the semantics of an enhanced object-oriented semantic network, where multiple instantiation, multiple specialization, and meta-classes are supported for both nodes and links.

The context of our work is the Semantic Index System (SIS) [14, 15, 16, 40]. In fact, the data model presented in this paper, is a (self-contained) part of the SIS data model. The SIS is targeted at supporting large descriptive knowledge structures in real applications. Typical applications include: cultural and scientific documentation systems, repository indexes for multimedia assets, engineering and software artifacts, laws and cases, organization structures, and other descriptive applications.

In all semantic network data models, specialization between classes is expressed through the *Isa* relation. However, as we demonstrated in [3], two different relations, *Isa* and *restriction isa* (*Risa*), are needed to express specialization between properties. The *Isa* relation between properties expresses inclusion of property extensions. The *Risa* relation is a stronger form of *Isa* and expresses property value refinement.

To get a feeling of *Risa*, consider the class *Art collector* that has a property *collects* with value in the class *Art object*, and the class *Painting collector* that has a property *collects* with value in the class *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 that stronger form of *Isa* that we call *Risa*. In our example, using *Risa* between the two *collects* properties, expresses that the property *collects* of *Art collector* restricted to *Painting collectors* takes values in *Painting*.

Inheritance is a well-known concept in the area of knowledge representation. However, it usually lacks formal definition and is defined procedurally. In [3], we formally defined property inheritance by employing the *Risa* relation.

The user defines objects and relations between objects through *declarations*. Specifically, there are two types of declarations: those that define *explicit objects* and those that define *explicit relations*. Explicit relations relate explicit objects through *In*, *Isa*, or *Risa* relations. A set of declarations that satisfies certain syntactical conditions, makes up a *program*.

Intuitively, a program corresponds to a semantic network with explicit information only.

Reasoning in our data model is done through a number of (built-in) *inference rules* that allow for derivations both at instance and schema level. In addition to the inference rules, a number of (built-in) *system constraints* are used for checking the validity of a program. Through the inference rules, new objects are derived, as well as new In, Isa, and $Risa$ relations between objects. We shall refer to objects and relations that are not explicit, as *derived objects* and *derived relations*, respectively.

In our data model, inherited properties are considered to be derived objects. This is important, as inherited properties *may not* have all the characteristics of the original properties. In particular, the inherited property may have a finer value domain than the original property. In fact, the value domain of an inherited property corresponds to the "intersection" of the value domains of several properties, including the original property. In addition, the inference rules relate inherited properties to other properties through Isa and $Risa$ relations. Such relations not only give useful information about the inherited properties but also refine the values of the inherited properties.

In this paper, we formally define the semantics of a program as follows: Each program P has a set of *models*. Intuitively, a model is a semantic network that satisfies the inference rules and the declarations in P. We define a partial ordering over the models of P and we prove that P has a least model, say M. If M satisfies the system constraints then we call it the *semantics* of P. A program with no semantics is called *invalid*. Intuitively, the semantics of a program P corresponds to an expanded semantic network which contains the explicit information defined in P, as well as additional information derived by the inference rules.

Thus the main contribution of this paper is to give a formal definition of the semantics of a semantic network supporting multiple instantiation, multiple specialization, and meta-classes. Moreover, the present paper provides a formal account for several semantic constructs introduced in an earlier paper [3].

The rest of the paper is organized as follows: Section 2 describes our view on objects, and defines the instantiation and specialization relations. Section 3 defines the derived objects of our data model. Sections 4 and 5 present the inference rules and the system constraints, respectively. Section 6 defines a program and the program semantics. Section 7 presents a comparison of our work with related works. Finally, section 8 contains concluding remarks. All proofs are given in the Appendix.

2. Our View on Objects

In our data model, objects are distinguished with respect to their nature into: *individuals*, *arrows*, and *hybrids*.

- **Individuals**: These are concrete or abstract entities of independent existence, such as the concrete entity *my-car*, and the abstract entity *Car*.

- **Arrows**: These are concrete or abstract properties/binary relationships[1] of objects. More precisely, an arrow a represents a property or a relationship from an object o to an object o'. The objects o, o' are called the *from* and *to* objects of the arrow a, and are denoted by $from(a)$ and $to(a)$, respectively.

[1]We do not make the distinction between property and binary relationship, as our approach is common to both.

Examples of arrows are: (i) the concrete relationship *produced* from *Opel* to *my-car*, and (ii) the abstract relationship *produced* from *Car-company* to *Car*.

- **Hybrids**: These are abstractions that refer collectively to objects of any kind.

  For example, the abstraction *Mathematical concept* is a hybrid object, as it refers collectively both to entities (e.g., *integer*) and relationships (e.g., *equal*).

In the present work, we consider only arrows whose *to* object is an individual or a hybrid. Yet, the *from* object of an arrow can be of any type. In particular, it can be another arrow.

Objects are also distinguished with respect to their concreteness, into *tokens* and *classes*.

- **Tokens:** These are concrete objects, such as the individual *my-car*, and the arrow *produced* from *Opel* to *my-car*.
- **Classes:** These are abstract objects, in the sense that they refer collectively to a set of objects that are considered similar in some respect. Classes that refer collectively also to classes are called *meta-classes*.

  Examples of classes are: (i) the individual *car*, and (ii) the arrow *produced* from *Car-company* to *Car*.

All hybrids are classes, as they refer collectively to a set of objects. In contrast, individuals and arrows can be tokens or classes.

Our view of objects as individuals, arrows, or hybrids on one hand, and as tokens or classes on the other, follows quite closely the structural part of the knowledge representation language Telos [30, 24].

The *extension* of a class c is the set of objects referred to collectively by c. We assume that the extension of an individual class is a set of individuals. The extension of an arrow class from a class c to a class c' is a set of arrows from objects in the extension of c to objects in the extension of c'. The extension of a hybrid is a set of individuals, arrows, or hybrids.

Objects can be related to classes through the *instance of* relation.

Definition 2.1. In relation
If an object o belongs to the extension of a class c then we say that o is an *instance of* c, and we denote it by $In(o, c)$. An object can be instance of zero, one, or more classes (multiple instantiation). $\diamond$

For an example, refer to Figure 1(a), where the arrow token *collects* from *art collector X* to *art object Y* is instance of the arrow class *collects* from *Art collector* to *Art object*.

2.1. Two forms of specialization: Isa and Risa

In this subsection, we define two forms of specialization: *Isa* and *Risa* (called *restriction isa*). The *Isa* relation relates pairs of classes and expresses inclusion of class extensions. The *Risa* relation relates pairs of arrow classes and expresses property value refinement.

Definition 2.2. Isa relation
Let o, o' be two classes. We distinguish three cases:
Case 1: o and o' are individual classes.

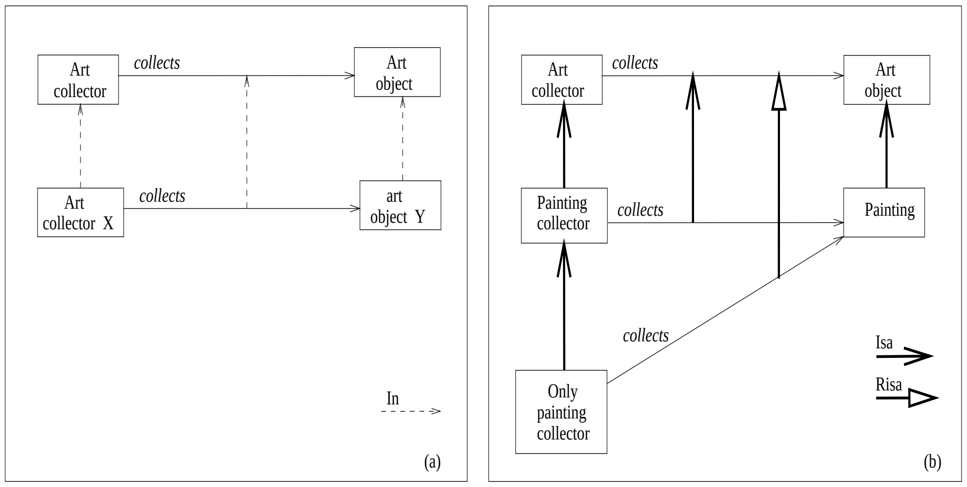

Figure 1. Example of relations between objects

We say that o is *subclass* of o', denoted by $Isa(o, o')$, if it holds that: for any x, if $In(x, o)$ then $In(x, o')$.
Case 2: o and o' are arrow classes.
We say that o is *subclass* of o', denoted by $Isa(o, o')$, if it holds that:
(i) $Isa(from(o), from(o'))$, (ii) $Isa(to(o), to(o'))$, and (iii) for any x, if $In(x, o)$ then $In(x, o')$.
Case 3: o is a class and o' is a hybrid.
We say that o is *subclass* of o', denoted by $Isa(o, o')$, if it holds that: for any x, if $In(x, o)$ then $In(x, o')$.
In all other cases, *Isa* is undefined.
A class can be subclass of zero, one, or more classes (multiple specialization). $\diamond$

See for example Figure 1(b), where the class *Painting collector* refers to art collectors that collect paintings (but may also collect other art objects). The arrow class *collects* from *Painting collector* to *Painting* is subclass of the arrow class *collects* from *Art collector* to *Art object*. This is because every painting collected by a painting collector is an art object collected by an art collector.

We now give the definition of the restriction isa relation.

Definition 2.3. 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 x, if $In(x, a')$ and $In(from(x), from(a))$ then $In(x, a)$. $\diamond$

For example, see Figure 1(b), where the class *Only painting collector* refers to art collectors that collect only paintings. The arrow *collects* of *Only painting collector* is a restriction subclass of the arrow *collects* of *Art collector*. This is because if an art collector collects an art object and the art collector happens to be an only-painting-collector, then the art object must be a painting.

3. Derived Objects

In our data model, objects are also characterized as *explicit* or *derived.* Explicit objects are declared by the user, whereas derived objects are derived by the system.

There are five kinds of derived objects: *meet classes*, *join classes*, *exact inherited arrows*, *to objects of exact inherited arrows*, and *approximate inherited arrows.*

Intuitively, meet classes are intersections of classes, join classes are unions of classes, and exact and approximate inherited arrows are properties inherited by subclasses. In particular, the *to* object of an approximate inherited arrow is a meet class. The exact inherited arrows are auxiliary derived objects that are introduced for the derivation of approximate inherited arrows.

3.1. Meet and Join Classes

In this subsection, we define the meet class and join class of a set of classes.

Definition 3.1. Meet class
Let s be a set of explicit individual or hybrid classes. We define the *meet class* of s, denoted by $meet(s)$, to be the class whose extension is the intersection of the extensions of the classes in s. $\diamond$

We distinguish two cases, depending on whether s has a least class w.r.t. Isa, or not:

Case 1: The set s has least class, i.e., there exists a class c in s such that c is subclass of each class in s. Then, it follows that $meet(s) = c$.

Case 2: The set s has no least class. Then, the system derives the class $meet(s)$ and the following Isa relations:
(i) For every class c in s, derive $Isa(meet(s), c)$.
(ii) For every class c such that c is subclass of each class in s, derive $Isa(c, meet(s))$.

Note that if $meet(s)$ is the least class of s (Case 1) then $meet(s)$ is an explicit object that already satisfies the Isa relations of (i) and (ii). Otherwise (Case 2), $meet(s)$ is a derived object.

For an example, refer to Figure 3. The set of classes $s = \{d', d_0, d_1\}$ has no least class, so $meet(\{d', d_0, d_1\})$ is a derived object. This derived object is subclass of d', d_0, and d_1.

Similarly to meet classes, we define the join classes.

Definition 3.2. Join class
Let s be a set of explicit individual or hybrid classes. We define the *join class* of s, denoted by $join(s)$, to be the class whose extension is the union of the extensions of the classes in s. $\diamond$

We distinguish two cases, depending on whether s has a greatest class w.r.t. Isa, or not:

Case 1: The set s has greatest class, i.e., there exists a class c in s such that each class in s is subclass of c. Then, it follows that $join(s) = c$.

Case 2: The set s has no greatest class. Then, the system derives the class $join(s)$ and the following Isa relations:
(i) For every class c in s, derive $Isa(c, join(s))$.
(ii) For every class c such that each class in s is subclass of c, derive $Isa(join(s), c)$.

3.2. 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'. We motivate our definition as follows (see Figure 2).

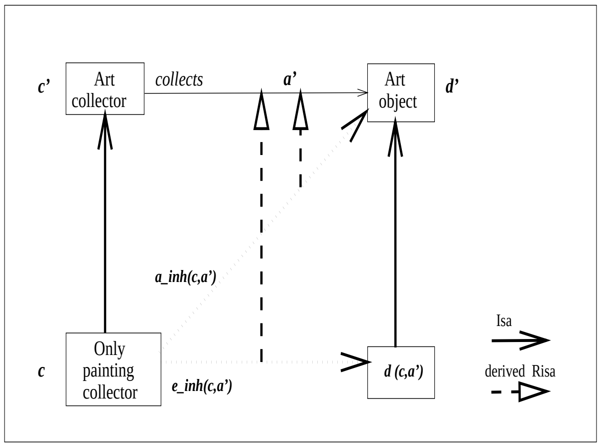

Figure 2. Exact and approximate arrow inheritance

The arrow a' is an arrow 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 an arrow with $from$ object c. It is this part that we shall call the arrow inherited by c from a'. Obviously, the extension E of this inherited arrow is the set of arrows in E' whose $from$ objects are instances of c. It now remains to determine the *to* object of this inherited arrow. Let $d(c, a')$ be the class referring to the *to* objects of the arrows in E. We take $d(c, a')$ to be the *to* object of the inherited arrow. We refer to this kind of inherited arrow, as the *exact inherited arrow*. Later in the section, we introduce the *approximate inherited arrow* that, in a sense, "approximates" the exact inherited arrow.

Definition 3.3. Exact 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 extension consists of the *to* objects of the arrows x such that $In(from(x), c)$ and $In(x, a')$ hold.

We define $e\_inh(c, a')$ to be the arrow class from c to $d(c, a')$ whose extension consists of the arrows x such that $In(from(x), c)$ and $In(x, a')$ hold. The arrow $e\_inh(c, a')$ is called the *exact arrow inherited* by c from a'. $\diamond$

Every exact inherited arrow $e\_inh(c, a')$ is a derived arrow. Additionally, $d(c, a')$ is a derived class. It easily follows that $e\_inh(c, a')$ is restriction subclass of a'. For example, in Figure 2, $e\_inh(c, a')$ is a derived arrow which 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. If we derive that $inh(c, a')$ is subclass of an arrow a, then we can derive that $d(c, a')$ is subclass of $to(a)$. In this way, although we do not know what exactly the class $d(c, a')$ is, we can derive a set of explicit classes which are superclasses of $d(c, a')$. This set is denoted by $cand\_cl(c, a')$ and is computed in subsection 4.3. We can say that the "intersection" of the classes in $cand\_cl(c, a')$ provides an "approximation" of the class $d(c, a')$.

Roughly speaking, as the classes in $cand\_cl(c, a')$ are explicit, their meet class is known (in contrast to the class $d(c, a')$ which is unknown). Therefore, we would like to define an inherited arrow that "approximates" $e\_inh(c, a')$ and its *to* object is the meet class of the classes in $cand\_cl(c, a')$. We denote this arrow by $a\_inh(c, a')$ and call it *approximate inherited arrow* or *inherited arrow*, for short. Obviously, $a\_inh(c, a')$ should be an arrow from c to $meet(cand\_cl(c, a'))$ whose extension is the same as that of $e\_inh(c, a')$. In [3], we prove that there is a unique arrow with these properties. Thus, $a\_inh(c, a')$ is well defined.

Definition 3.4. Inherited arrow
Let a' be an explicit arrow class from a class c' to a class d' and let c be a subclass of c'. We define $a\_inh(c, a')$ to be the arrow class from c to $meet(cand\_cl(c, a'))$ whose extension consists of the arrows x such that $In(from(x), c)$ and $In(x, a')$ hold. The arrow $a\_inh(c, a')$ is called the *approximate arrow inherited* by c from a', or *arrow inherited* by c from a', for short. $\diamond$

Similarly to $e\_inh(c, a')$, $a\_inh(c, a')$ is a derived arrow which is restriction subclass of a'.

Examples
In Figure 2, we have $cand\_cl(c, a') = \{Art\ object\}$. Therefore, as *Art object* is the least object of $cand\_cl(c, a')$, it follows that the *to* object of $a\_inh(c, a')$ is *Art object*. Note that $a\_inh(c, a')$ is a derived arrow which is restriction subclass of a'.

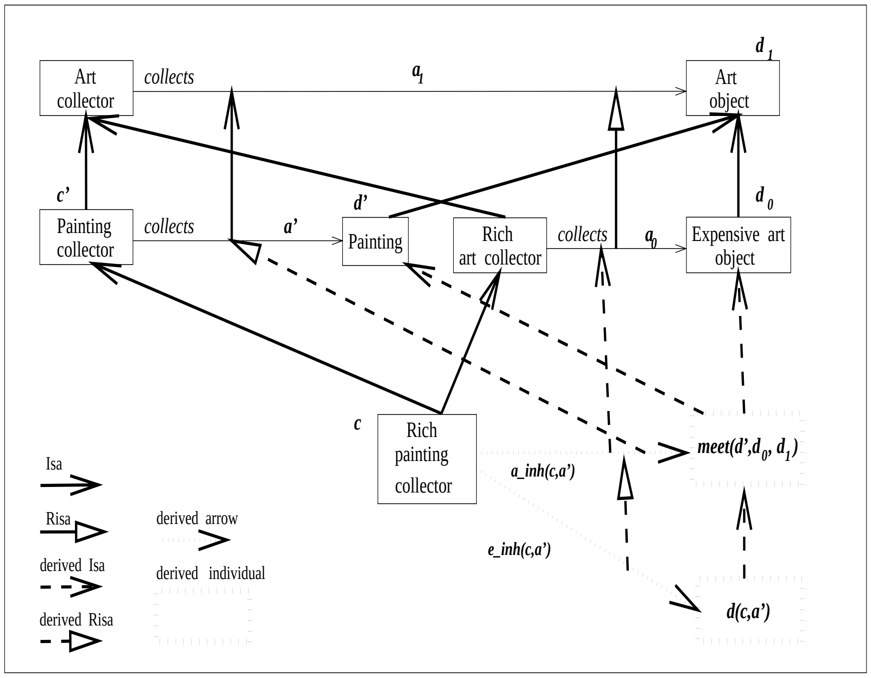

Figure 3. Example of arrow inheritance

For another example, refer to Figure 3, where we have that $cand\_cl(c, a_0) = \{d', d_0, d_1\}$. The *to* object of $a\_inh(c, a')$ is the derived class $meet(cand\_cl(c, a'))$.

An extended discussion and illustrative examples regarding inherited arrows and their usefulness can be found in [3].

4. Inference Rules

As we mentioned earlier, the In, Isa, and $Risa$ relations are either declared or derived. Relation derivations are performed using certain inference rules. Additionally, inference rules are used for generating derived objects. All inference rules of our model are sound. The soundness of the Isa, Risa, and Exact Inheritance Rules is proved in [3]. The soundness of the rest of the inference rules is proved, similarly.

O: set of objects	*A: set of arrows*	*IC: set of individual classes*	*EA: set of explicit arrows*
I: set of individuals	*T: set of tokens*	*AC: set of arrow classes*	*EAC: set of explicit arrow classes*
H: set of hybrids	*C: set of classes*	*E: set of explicit objects*	*L: powerset of explicit individual and hybrid classes*

Figure 4. Notations of object sets

We use O to denote the set of all objects. We use I, H, A, T, C, and E to denote the sets of individuals, hybrids, arrows, tokens, classes, and explicit objects, respectively. We use $IC = I \cap C$, $AC = A \cap C$, $EA = E \cap A$, and $EAC = E \cap AC$ to denote the sets of individual classes, arrow classes, explicit arrows, and explicit arrow classes, respectively. We use $\mathcal{L} = \mathcal{P}((IC \cup H) \cap E)$ to denote the powerset of explicit individual and hybrid classes. These notations are shown in Figure 4.

4.1. Isa and Risa Rules

The *Isa Rules* are used for deriving (i) new Isa relations based on given Isa relations, and (ii) new In relations based on given In and Isa relations.

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

The *Risa Rules* are used for deriving (i) new Isa and $Risa$ relations based on given Isa and $Risa$ relations, and (ii) new In relations (between arrows) based on given In and $Risa$ relations.

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

Risa Rules 1 and 2 reflect the definition of *Risa*. We refer to [3], for illustrative examples and discussions regarding the Risa Rules.

4.2. Meet and Join Rules

The *Meet Rules* derive meet classes and relate them to other objects. Recall that $\mathcal{L}$ is the powerset of explicit individual and hybrid classes and $meet : \mathcal{L} \rightarrow C$.

MEET RULES

Rule 1: $\forall\, s \in \mathcal{L},\, c \in IC \cup H, \quad c \in s \wedge (\forall\, c' \in s, Isa(c, c')) \Rightarrow meet(s) = c$

Rule 2: $\forall\, s \in \mathcal{L},\, c \in IC \cup H, \quad c \in s \Rightarrow Isa(meet(s), c)$

Rule 3: $\forall\, s \in \mathcal{L},\, c \in IC \cup H, \quad (\forall\, c' \in s, Isa(c, c')) \Rightarrow Isa(c, meet(s))$

Rule 4: $\forall\, s \in \mathcal{L},\, o \in O, \quad (\forall\, c' \in s, In(o, c')) \Rightarrow In(o, meet(s))$

Rule 5: $\forall\, s, s' \in \mathcal{L}, \quad Isa(meet(s), meet(s')) \wedge Isa(meet(s'), meet(s)) \Rightarrow meet(s) = meet(s')$

Note that Meet Rules 2 and 3 derive *Isa* relations for $meet(s)$, where $s \in \mathcal{L}$. From Meet Rule 1, it follows that if c is the least class of s then $meet(s)$ is the explicit class c. Additionally, it will follow from the definition of the program semantics that the inverse is also true. Specifically, if $meet(s)$ is an explicit class c then c is the least class of s. Therefore, it follows that if $meet(s)$ is an explicit class c then the *Isa* relations derived by Meet Rules 2 and 3, are already holding for c. Thus, these rules do not derive new *Isa* relations for explicit classes.

Note that the inverse of Meet Rule 3 follows from Meet Rule 2 and Isa Rule 2. Additionally, the inverse of Meet Rule 4 follows from Meet Rule 2 and Isa Rule 3.

Consider a set $s \in \mathcal{L}$ such that $o, o' \in s$ and o is subclass of o'. It holds that $meet(s){=}meet(s\text{ - }\{o'\})$, as expected. This result is an easy consequence of Meet Rules 2,3,5 and Isa Rule 2. For example, in Figure 3, it holds that $meet(d', d_0, d_1) = meet(d', d_0)$.

Similarly to the Meet Rules, the Join Rules derive join classes and relate them to other objects.

JOIN RULES

Rule 1: $\forall\, s \in \mathcal{L}, c \in IC \cup H, \quad c \in s \,\wedge\, (\forall\, c' \in s, Isa(c', c)) \;\Rightarrow\; join(s) = c$

Rule 2: $\forall\, s \in \mathcal{L}, c \in IC \cup H, \quad c \in s \;\Rightarrow\; Isa(c, join(s))$

Rule 3: $\forall\, s \in \mathcal{L}, c \in IC \cup H, \quad (\forall\, c' \in s, Isa(c', c)) \;\Rightarrow\; Isa(join(s), c)$

Rule 4: $\forall\, s, s' \in \mathcal{L}, \quad Isa(join(s), join(s')) \,\wedge\, Isa(join(s'), join(s)) \;\Rightarrow\; join(s) = join(s')$

4.3. Inheritance Rules

In this subsection, we present two sets of inference rules: *Exact Inheritance Rules* and *Approximate Inheritance Rules*. The *Exact Inheritance Rules* derive exact inherited arrows[2] and relate them to other arrows. Based on derivations for exact inherited arrows, the *to* object of inherited arrows is computed.

EXACT INHERITANCE RULES

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

$$Isa(c, from(a')) \;\Leftrightarrow \exists\; e\_inh(c, a') \in AC$$
$$\exists\; e\_inh(c, a') \in AC \Rightarrow from(e\_inh(c, a')) = c$$

Rule 2: $\forall\, a' \in EAC, \;\; c \in C, \quad Risa(e\_inh(c, a'), \; a')$

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

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

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

Rule 5: $\forall\, a' \in EAC, d \in C \cap E, \;\; c \in C, \quad Isa(to(e\_inh(c, a')), \; d) \Rightarrow \; d \in cand\_cl(c, a')$

Rule 6: $\forall\, a' \in EAC, \;\; c \in C, \;\; x \in A, \quad In(x, e\_inh(c, a')) \;\Rightarrow\; In(to(x), to(e\_inh(c, a')))$

Rule 7: $\forall\, a', a'' \in EAC, \;\; c \in C,$

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

Exact Inheritance Rule 4, refines the class $to(e\_inh(c, a'))$. That is, it relates the *to* object of $e\_inh(c, a')$ to its superclasses through the Isa relationship. Exact Inheritance Rule 5, puts the superclasses of $to(e\_inh(c, a'))$ into the *candidate class set* for c and a' (i.e., $cand\_cl(c, a')$). As this rule is the only rule that defines $cand\_cl(c, a')$, the set $cand\_cl(c, a')$ includes all superclasses of $to(e\_inh(c, a'))$.

For an example of Exact Inheritance Rule 3, refer to Figure 3. First, observe that arrow $e\_inh(c, a')$ is a restriction subclass of a' (Exact Inheritance Rule 2). Additionally, observe

[2]Note that $e\_inh : C \times EAC \rightarrow AC$ is a partial function. In the inference rules, the expression $\exists\; e\_inh(c, a') \in AC$ evaluates to *true*, if $e\_inh(c, a')$ is defined (in other words, if the arrow $e\_inh(c, a')$ is derived).

that a' is subclass of a_1. Thus, from *Isa* Rule 2 (transitivity), we derive that $e\_inh(c, a')$ 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 Exact Inheritance Rule 3 satisfied. Thus, we obtain that $e\_inh(c, a')$ is subclass of a_0.

For an example of Exact Inheritance Rule 4, we continue with our previous example (Figure 3). As $e\_inh(c, a')$ is subclass of a', the *to* object of $e\_inh(c, a')$ (denoted by $d(c, a')$) is subclass of d'. As $e\_inh(c, a')$ is subclass of a_1, $d(c, a')$ is subclass of d_1. Additionally, as $e\_inh(c, a')$ is subclass of a_0, $d(c, a')$ is subclass of d_0. As no other superclass of $d(c, a')$ can be derived, it follows that $cand\_cl(c, a') = \{d', d_0, d_1\}$. Additionally, it follows from Meet Rule 3 that $d(c, a')$ is subclass of $meet(\{d', d_0, d_1\})$.

The *Approximate Inheritance Rules* derive inherited arrows and relate them to other arrows.

APPROXIMATE INHERITANCE RULES

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

$$Isa(c, from(a')) \Leftrightarrow \exists\, a\_inh(c, a') \in AC$$
$$\exists\, a\_inh(c, a') \in AC \Rightarrow from(a\_inh(c, a')) = c$$

Rule 2: $\forall\, a' \in EAC,\ \ c \in C,\quad Risa(a\_inh(c, a'),\ a')$

Rule 3: $\forall\, a', a'' \in EAC,\ \ c \in C,$

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

Rule 4: $\forall\, a' \in EAC,\ \ c \in C,\quad \exists\, a\_inh(c, a') \in AC \Rightarrow to(a\_inh(c, a')) = meet(cand\_cl(c, a'))$

For example, in Figure 3, $a\_inh(c, a')$ is restriction subclass of a' and the *to* object of $a\_inh(c, a')$ is $meet(\{d', d_0, d_1\})$.

5. System Constraints

The following *system constraints* guarantee the validity of the information base. Though suitable forms of these constraints could be used as inference rules, we have decided to use them as constraints for checking the validity of explicitly declared information.

For example, the inference rule corresponding to the *Isa Antisymmetry Constraint* (below) is the following: If c is a subclass of c' and c' is a subclass of c then (infer that) c and c' coincide. Obviously, to infer that two explicitly declared classes coincide may lead to wrong conclusions. Thus, in this case, we thought it more appropriate to indicate the problem to the user, rather than inferring that the two classes coincide.

SYSTEM CONSTRAINTS

ISA CONSTRAINTS

1. *Isa Domain Constraint:* $Isa \subseteq C \times C$
2. *Isa Antisymmetry Constraint:* $\forall\, a, b \in O,\quad Isa(a,b) \wedge Isa(b,a) \Rightarrow a = b$
3. *Arrow Isa Constraint:* $\forall\, a, a' \in A,\quad Isa(a,a') \Rightarrow Isa(from(a), from(a')) \wedge Isa(to(a), to(a'))$

IN CONSTRAINTS

1. *In Domain Constraint:* $In \subseteq O \times C$
2. *Arrow In Constraint:* $\forall\, x, a \in A,\quad In(x,a) \Rightarrow In(from(x), from(a)) \wedge In(to(x), to(a))$

CONCRETENESS CONSTRAINTS

1. *Concreteness Constraint:* $T \cap C = \emptyset$

The Arrow In and Isa Constraints are used for typechecking (as we shall see in section 7).

6. Semantics of Declaration Programs

We first define the notion of program and then give its formal semantics. Roughly speaking, the semantics of a program is expressed by means of what we call a *model.* We prove that every program P has a least model, say M. If M satisfies the system constraints then M is considered to represent the *semantics* of P. Otherwise, P is considered to be *invalid.*

6.1. Declaration Programs

The user interacts with the system by declaring the objects of interest, as well as the relationships between these objects. A set of declarations is what we call a "program". A program is "interpreted" by the system in order to build the information base (expanded semantic network).

To name the objects of interest and their relationships, the user uses a set of *naming symbols*, denoted by N.

Intuitively, each explicit individual and hybrid has a *name* in N, and each explicit arrow has a *label* in N. The name of an explicit arrow a is formed by the name of $from(a)$ and the label of a. Specifically, if f is the name of $from(a)$ and l is the label of a then the name of a is f/l. For example, if an arrow a has label *produced* and the name of its *from* object is *Car-company* then the name of a is *Car-company/produced.*

Definition 6.1. Object names
We define the arrow names A_{nam} and object names O_{nam}, as follows:

$$A_{nam} = \{n/l_1/.../l_k \mid n, l_1, ..., l_k \in N\}$$
$$O_{nam} = N \cup A_{nam} \quad \diamond$$

With names, the user can refer only to explicit objects. In order to refer to both explicit and derived objects, the user needs references. In fact, references were introduced for the external identification of derived objects. Each object has an associated set of references.

Intuitively, references are formed from (explicitly defined) names. The name of an explicit object o is always a reference to o.

Let s be a set of explicit classes. References to $meet(s)$ and $join(s)$ are formed from the names of objects in s. For an example, refer to Figure 3. Note that *Painting, Expensive_art_object*, and *Art_object* are the names of the objects d', d_0, and d_1, respectively. Therefore, a reference to $meet(\{d', d_0, d_1\})$ is $meet(\{$*Painting, Expensive_art_object, Art_object*$\})$.

A reference to $e\_inh(c, a')$ is formed from a reference to c and the name of a'. Specifically, if r is a reference to c and n is the name of a' then $e\_inh(r, n)$ is a reference to $e\_inh(c, a')$. Similarly, if r is a reference to c and n is the name of a' then $a\_inh(r, n)$ is a reference to $a\_inh(c, a')$.

Arrow references A_{ref} and object references O_{ref} are defined in steps as follows:

Definition 6.2. Object references
We first define the step i arrow references A^i_{ref} and object references O^i_{ref}, as follows:

for $i = 0$:

$$A^0_{ref} = A_{nam}, \quad O^0_{ref} = O_{nam} \cup meet(\mathcal{P}(N)) \cup join(\mathcal{P}(N))$$

for $i > 0$:

$$\begin{aligned} A^i_{ref} &= \{e\_inh(r, n) \mid r \in O^{i-1}_{ref},\ n \in A_{nam}\} \cup \\ &\quad\ \{a\_inh(r, n) \mid r \in O^{i-1}_{ref},\ n \in A_{nam}\} \\ O^i_{ref} &= A^i_{ref} \cup \{d(r, n) \mid r \in O^{i-1}_{ref},\ n \in A_{nam}\} \end{aligned}$$

We now define the arrow references A_{ref} and the object references O_{ref}:

$$A_{ref} = \bigcup_{i \geq 0} A^i_{ref}, \quad O_{ref} = \bigcup_{i \geq 0} O^i_{ref} \quad \diamond$$

Intuitively, the object references of step 0 are the object names and the references to meet classes. The arrow references of step i are the references of exact and approximate arrows inherited by a class c from an arrow a, where the reference to c was constructed at step $i - 1$. The object references of step i are (i) the arrow references of step i, and (ii) the references to the *to* objects of the arrows $e\_inh(c, a')$, where the reference to c was constructed at step $i - 1$.

The user declares the objects of interest and their relationships through the following *declarations*.

Definition 6.3. Declarations
A *declaration* is one of the following expressions:

- $indiv(n)$, where $n \in N$. It declares an individual with name n.
- $hybrid(n)$, where $n \in N$. It declares a hybrid with name n.
- $arrow(f, l, t)$, where $f \in O_{nam},\ l, t \in N$. It declares an arrow with label l that goes from an object with name f to an individual or hybrid with name t.
- $token(n)$, where $n \in O_{nam}$. It declares that the object with name n is a token.
- $class(n)$, where $n \in O_{nam}$. It declares that the object with name n is a class.

- $in(n, n')$, where $n, n' \in O_{nam}$. It declares an object with name n to be instance of an object with name n'.
- $isa(n, n')$, where $n, n' \in O_{nam}$. It declares an object with name n to be subclass of an object with name n'.
- $risa(n, n')$, where $n, n' \in A_{nam}$. It declares an arrow with name n to be restriction subclass of an arrow with name n'. $\diamond$

We now give the definition of declaration program.

Definition 6.4. Declaration program
We call *declaration program* or simply *program*, any set of declarations such that the following conditions hold:

1. There is no pair of declarations $indiv(n)$, $hybrid(n)$.
2. There is no pair of declarations $arrow(f, l, t)$ and $arrow(f, l, t')$, for $t \neq t'$.
3. For every name n that appears in P,
   - if $n \in N$ then there is a declaration $indiv(n)$ or $hybrid(n)$,
   - if $n = n'/l$, where $l \in N$, then there is a declaration $arrow(n', l, t)$. $\diamond$

All conditions above are syntactical. Condition 1 expresses that there cannot be an individual and a hybrid with the same name. Condition 2 expresses that arrow labels should be unique within their $from$ object. Condition 3 expresses that if a name n appears in a program then an object with name n should have been declared.

In the SIS system [14], we have developed a higher-level data entry language. Commands in this higher-level language are translated into a set of declarations which in case they satisfy the above conditions, they form a declaration program.

6.2. Semantic Structures

The system "interprets" a program in order to build the information base of the application. To do this, the system needs to create objects, associate names and references to these objects, and build relations between these objects. The intended interpretation of a program is defined by means of what we shall call a semantic structure.

Definition 6.5. Semantic structure
A *semantic structure* is defined by a tuple: $S =< O, from, to, Rel, DerObj, Ref >$,
where: $Rel =< In, Isa, Risa >$, $DerObj =< meet, e\_inh, a\_inh >$, and
$Ref =< name, label, obj >$, such that:

- O is the set of *objects* of S. O contains three mutually disjoint subsets I, A, H, whose elements are called *individuals*, *arrows*, and *hybrids* of S, respectively. Additionally, O contains the sets T, C, and E whose elements are called *tokens*, *classes*, and *explicit objects* of S, respectively. In what follows, we use the notations shown in Figure 4.
- $from : A \to O$ and $to : A \to I \cup H$ are total functions that associate each arrow with its $from$ and to objects.

- $In \subseteq (I \times I) \;\cup\; (A \times A) \;\cup\; (O \times H)$ is a relation expressing the instance relation between objects.
- $Isa \subseteq (I \times I) \;\cup\; (A \times A) \;\cup\; (O \times H)$ is a relation expressing the subclass relation between objects. Isa must satisfy the *Isa Rules* (see subsection 4.1).
- $Risa \subseteq A \times A$ is a relation expressing the restriction subclass relationship between arrows. $Risa$ must satisfy the *Risa Rules* (see subsection 4.1).
- $meet : \mathcal{L} \rightarrow C$ is a total function that associates a set $s \in \mathcal{L}$ with the class $meet(s)$. The function $meet$ must satisfy the *Meet Rules* (see subsection 4.2).
- $e\_inh : C \times EAC \rightarrow AC$ is a partial function that associates a class c and an explicit arrow class a' with the exact arrow inherited by c from a'. The function e_inh must satisfy the *Exact Inheritance Rules* (see subsection 4.3).
- $a\_inh : C \times EAC \rightarrow AC$ is a partial function that associates a class c and an explicit arrow class a' with the arrow inherited by c from a'. The function a_inh must satisfy the *Approximate Inheritance Rules* (see subsection 4.3).
- $name : E \rightarrow O_{nam}$ is a total, one-to-one function that associates an explicit object with its name. The function $name$ must satisfy the *Name Rules* (see further on).
- $label : EA \rightarrow ARR$ is a total function that associates an explicit arrow with its label.
- $obj : O_{ref} \rightarrow O$ is a partial, surjective function that associates object references to objects. The function obj must satisfy the *Reference Rules* (see further on).

NAME RULES

Rule 1: $\forall\, a \in EA,\ f \in O_{nam},\ l \in ARR, \quad name(from(a)) = f \ \wedge\ label(a) = l \ \Rightarrow\ name(a) = f/l$

REFERENCE RULES

Rule 1: $\forall\, o \in E, n \in O_{nam} \quad name(o) = n \ \Rightarrow\ obj(n) = o$

Rule 2: $\forall\, s \in \mathcal{L}, \quad obj(meet(\{name(o) \mid o \in s\}) = meet(s)$
$\forall\, s \in \mathcal{L}, \quad obj(join(\{name(o) \mid o \in s\}) = join(s)$

Rule 3: $\forall\, a' \in EAC,\ c \in C,\ r \in O_{ref},\ n \in A_{nam},$

$$\exists\, e\_inh(c,a') \in AC \ \wedge\ obj(r) = c \ \wedge\ name(a') = n \ \Rightarrow$$
$$obj(e\_inh(r,n)) = e\_inh(c,a') \ \wedge\ obj(d(r,n)) = to(e\_inh(c,a'))$$

Rule 4: $\forall\, a' \in EAC,\ c \in C,\ r \in O_{ref},\ n \in A_{nam},$

$$\exists\, a\_inh(c,a') \in AC \ \wedge\ obj(r) = c \ \wedge\ name(a') = n \ \Rightarrow\ obj(a\_inh(r,n)) = a\_inh(c,a')$$

We define the *reference set* of an object $o \in O$ to be the set of references of o. Specifically, $ref(o) = \{r \in O_{ref} \mid obj(r) = o\}$. As obj is surjective, $ref(o) \neq \{\}$.

It is possible that, though two structures are different, they represent the same semantic network. In this case, we say that S and S' are *equivalent*.

Definition 6.6. Structure equivalence
Let S, S' be two structures with sets of objects O, O', respectively. We say that S and S' are *equivalent*, denoted by $S \equiv S'$, iff there is a bijective mapping $\mathcal{F} : O \rightarrow O'$ such that the structure that results after replacing every object $o \in O$ with $\mathcal{F}(o)$ is identical to S'. $\diamond$

It is easy to see that $\equiv$ is an equivalence relation over structures.

6.3. Program Semantics

In this subsection we define the models, as well as the semantics of a program.

Definition 6.7. Satisfaction of declarations
Let S be a semantic structure and let D be a declaration. We define S to *satisfy* D, denoted by $S \models D$, as follows:

- $S \models indiv(n)$ iff there is $i \in I \cap E$ such that $name(i) = n$.
- $S \models hybrid(n)$ iff there is $h \in H \cap E$ such that $name(h) = n$.
- $S \models arrow(f, l, t)$ iff there is $a \in A \cap E$ such that $name(from(a)) = f$, $label(a) = l$, and $name(to(a)) = t$.
- $S \models token(n)$ iff there is $o \in T \cap E$ such that $name(o) = n$.
- $S \models class(n)$ iff there is $o \in C \cap E$ such that $name(o) = n$.
- $S \models in(n, n')$ iff $In(obj(n), obj(n'))$.
- $S \models isa(n, n')$ iff $Isa(obj(n), obj(n'))$.
- $S \models risa(n, n')$ iff $Risa(obj(n), obj(n'))$. $\diamond$

Definition 6.8. Model of a program
Let P be a program and let M be a semantic structure. We say that M is a *model* of P if M satisfies every declaration of P. $\diamond$

We now introduce an ordering over the models of a program that allows to compare them with respect to their information content. We will then prove that every program P has a least model.

Definition 6.9. Model ordering
Let M and M' be two models[3] of P. We say that M is less than or equal to M', denoted $M \leq M'$, iff there is a mapping $\mathcal{F} : O \rightarrow O'$ such that

[3] Symbols of structure components with a prime, such as O', $name'$, denote components of M'.

1. $\forall\, o \in E, \quad name(o) = name'(\mathcal{F}(o))$.
2. $\forall\, o \in O, \quad ref(o) \subseteq ref'(\mathcal{F}(o))$.
3. $\forall\, c, a' \in O$, if $cand\_cl(c, a')$ is defined then $\{\mathcal{F}(x) \mid x \in cand\_cl(c, a')\} \subseteq cand\_cl'(\mathcal{F}(c), \mathcal{F}(a'))$.
4. $\forall\, o, o' \in O,\ Rel \in \{In, Isa, Risa\}$, if $Rel(o, o')$ then $Rel'(\mathcal{F}(o), \mathcal{F}(o'))$. $\diamond$

Proposition 6.1. *Let P be a program. The relation "$\leq$" is a partial ordering over the models of P (up to model equivalence).*

The following theorem is the main theorem of the paper.

Theorem 6.1. *Every program P has a least model.*

We shall call this least model the *semantics* of the program if it satisfies the *system constraints* (see Section 5).

Definition 6.10. Semantics of a program
Let P be a program and let M be the least model of P. We say that P is a *valid* program and M is the *semantics* of P if M satisfies the system constraints. Otherwise, we say that P is *invalid*. $\diamond$

In the rest of the section, we consider only valid programs P. Moreover, symbols of structure components, such as O, e_inh, refer to the semantics of P.

Proposition 6.2. *Let P be a valid program. Then, for each $o \in E$, it holds that either $o \in I$, or $o \in A$, or $o \in H$.*

As we have seen in Section 2, each object is either a token or a class. In the semantics of a program, however, an object o may be neither a token (i.e., $o \notin T$) nor a class (i.e., $o \notin C$). This is possible if the object is neither declared as instance of *Token* or *Class* (through the program) nor this is derived. Note that, due to Concreteness Constraint, an object cannot be both token and class.

In the rest of the section, we prove several properties of exact and approximate inherited arrows. The following proposition says that each exact inherited arrow is restriction subclass of its corresponding inherited arrow.

Proposition 6.3. *Let P be a valid program. Let a' be an arrow class and let c be a subclass of $from(a')$. Then, the arrow $e\_inh(c, a')$ is restriction subclass of $a\_inh(c, a')$.*

A consequence of the above proposition is that the arrows $e\_inh(c, a')$ and $a\_inh(c, a')$ share the same instances. This is derived as follows: From the above proposition, Risa Rule 1, and Isa Rule 3, it follows that every instance of $e\_inh(c, a')$ is also instance of $a\_inh(c, a')$. Note that the two arrows have the same *from* object. Therefore, from Risa Rule 2, every instance of $a\_inh(c, a')$ is also instance of $e\_inh(c, a')$.

The following proposition expresses that if an arrow a_0 is restriction subclass of an arrow a_1 then $e\_inh(c, a_0)$ coincides with $e\_inh(c, a_1)$, and $a\_inh(c, a_0)$ coincides with $a\_inh(c, a_1)$.

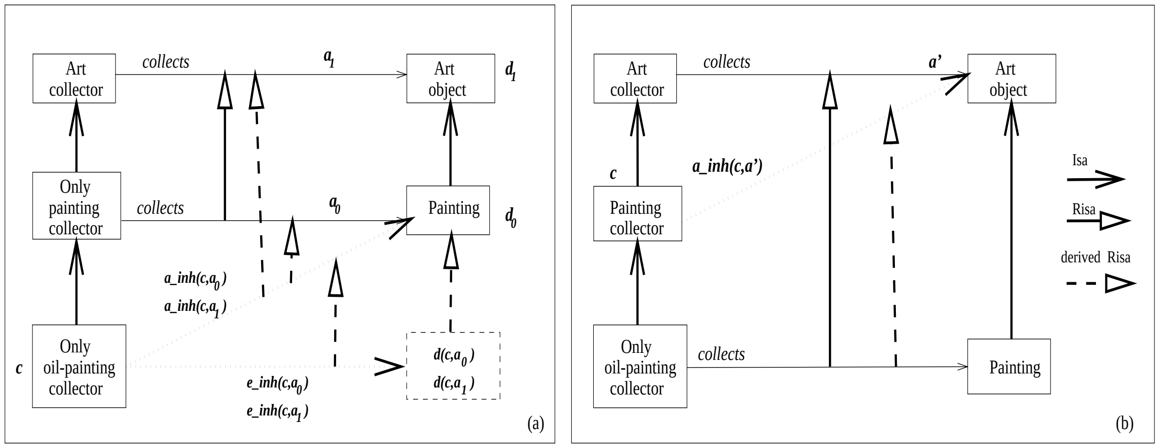

Figure 5. (a) Example of inherited arrows that coincide, (b) Example of an explicit arrow being subclass of an inherited arrow

Proposition 6.4. *Let P be a valid program. Let a_0, a_1 be explicit arrow classes and let c be a class. If $Risa(a_0, a_1)$ and $Isa(c, from(a_0))$ then $e\_inh(c, a_0) = e\_inh(c, a_1)$ and $a\_inh(c, a_0) = a\_inh(c, a_1)$.*

For an example, refer to Figure 5(a). Intuitively, the arrow a_0 refines the value of the arrow a_1. This refinement is expressed by the *Risa* relation. Thus, the value of the arrow inherited by c from a_1 is, in general, the same or finer than the value of a_0 (in this example, it is $to(a_0)$). This corresponds to property refinement[4] in object-oriented data models.

The following proposition gives an interesting property of inherited arrows. Specifically, it expresses that if the 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 6.5. *Let P be a valid program. Let a' be an explicit class and let c be a subclass of $from(a')$. If the arrow $a\_inh(c, a')$ is restriction subclass of an explicit arrow a then $e\_inh(c, a') = e\_inh(c, a)$ and $a\_inh(c, a') = a\_inh(c, a)$.*

The following proposition expresses that there are no explicit objects which are subclasses of the *to* objects of exact inherited arrows.

Proposition 6.6. *Let P be a valid program. There is no explicit class or meet class d such that d is subclass of the to object of an exact inherited arrow.*

Note that the *to* object of an explicit arrow is an explicit object and the *to* object of an inherited arrow is a meet class. Therefore, from the above proposition and the Arrow Isa Constraint, it follows that no explicit arrow or inherited arrow can be subclass of an exact inherited arrow. In contrast, inherited arrows can have subclasses which are explicit arrows or inherited arrows. An example is shown in Figure 5(b).

[4]Also called *type refinement*.

6.4. Discussion

We want to emphasize that though the formalization of *Isa*, *Risa*, and *Exact Inheritance* could be done using first order logic, the same is not true for *Approximate Inheritance.* This is because the Approximate Inheritance Rule 4 that determines the *to* object of an approximate inherited arrow, is not monotonic. Obviously, as additional information augments the candidate class set $cand\_cl(c, a')$, the *to* object of the approximate inherited arrow $a\_inh(c, a')$ is modified. In other words, $a\_inh(c, a')$ is computed based on the information currently in the information base, and it may be changed as new information is added. Thus, the computation of $a\_inh(c, a')$ is based on an implicit closed world assumption.

Our inference rules are sound but not complete. For example, there are *Isa* relations derived from the following (sound) inference rule:

$$\forall\ a_1, ..., a_n \in AC,$$

$$Isa(a_1, a_2)\ \wedge\ Risa(a_3, a_2)\ \wedge\ Isa(a_3, a_4)\ \wedge\ Risa(a_5, a_4)\ \wedge\ ...\ \wedge\ Isa(a_{n-2}, a_{n-1})\ \wedge\ Risa(a_n, a_{n-1})\ \wedge$$
$$(\forall i \in \{3, ..., n\},\ Isa(from(a_1), from(a_i)))\ \wedge\ Isa(to(a_1), to(a_n))$$
$$\Rightarrow\ Isa(a_1, a_n)$$

that cannot be derived through our inference rules. However, we claim that our inference rules can give a large number of interesting schema derivations, as shown in [3]. Note that Risa Rule 5 is a special case of the above inference rule ($n = 3$). We suspect that completeness is achieved if Risa Rule 5 is replaced by the above inference rule.

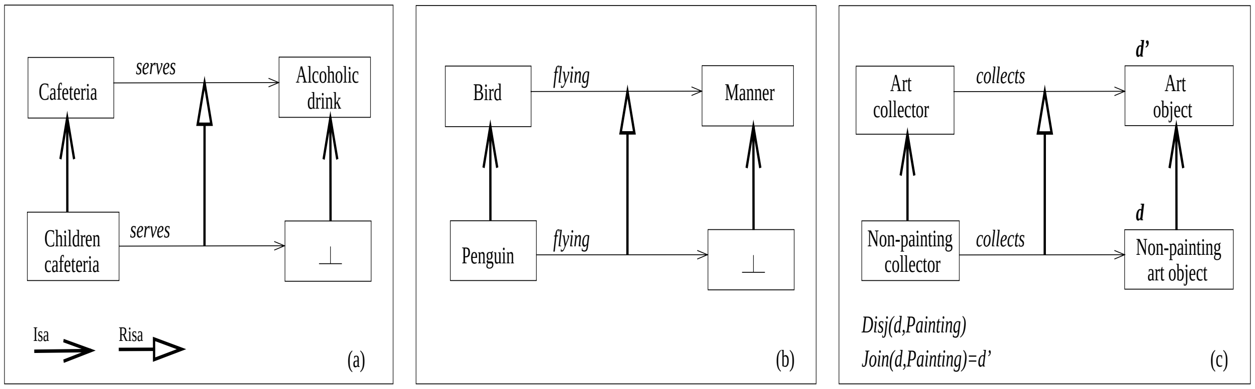

Figure 6. Expressing exceptions and negative information through our model

Our model can also express exceptions and negative information. For example, it can express that cafeterias for children do not serve alcoholic drinks, or that penguins do not fly, through the declarations shown in Figures 6(a) and 6(b). This is because there cannot be an instance of the arrow class *serves* of *Cafeteria* whose *from* object is instance of *Children cafeteria.* Indeed, if there is such an arrow x then x should also be instance of the arrow class *serves* of *Children cafeteria.* However, this is impossible because $to(x)$ cannot be instance of the class $\perp$[5]. For similar reasons, there cannot be an instance of the arrow class *flying* of *Bird* whose *from* object is instance of *Penguin.*

As another example, our model can express that a non-painting collector is an art collector that collects art objects other than paintings, through the declarations shown in

[5]The class $\perp$, called *empty class,* is the class whose extension is empty.

Figure 6(c). In the figure, the declaration $Disj(d, Painting)$ expresses that the extensions of the classes *Non-painting art object* and *Painting* are disjoint. Additionally, the declaration $join(d, Painting) = d'$ expresses that the extension of the class *Art object* is the union of the extensions of the classes *Non-painting art object* and *Painting*.

7. Comparison with Related Work

Specialization hierarchies and inheritance play an important role in knowledge representation systems based on semantic networks and frames [23, 34], as well as in object-oriented and some extensible database systems [27, 13]. Yet, many of these systems define their semantics and, in particular, property inheritance, in a procedural way. A detailed comparison between our approach to inheritance and that of several systems, such as ORION [5], O_2 [17], ODE [2], POSTGRES [35, 37], EXODUS [10], is given in [3].

In this section, we review systems that define their semantics based on logic. We first establish a common framework and vocabulary for comparing our data model with related ones.

Let p be a property of a class c with value d. We say that p is a *typing property* of c, if c refers collectively to properties of instances of c with value in d. The typing property p not only expresses information about the class c and its instances but is also used for checking the validity of a program through the *Typing Constraint*: the properties that are referred collectively by p should have value in d.

The set of typing properties of a class c is called the *type* of c [11]. A type T is a *subtype* of a type T' iff T supports all properties of T' with the same or more refined value domain (property refinement). T may have additional properties. If a class c is subclass of a class c' then the type of c should be subtype of the type of c' (*Subtyping Constraint*). Checking of the Typing and Subtyping Constraints is usually referred to as *typechecking.*

The typing properties of a class c are either *local* in c or *inherited* from superclasses of c. Thus, the type of a class depends on the inheritance semantics of the particular data model. In fact, as we show in [3], not all data models satisfy the Subtyping Constraint.

In our data model, typing properties are modeled by arrow classes and, conversely every arrow class models a typing property. The local typing properties of c are the explicit arrow classes of c which are not restriction subclasses of any other arrow class. The inherited typing properties of c are the (approximate) arrows inherited by c.

In our data model, we indicate that a property x is referred to by a typing property p by declaring that x is an instance of p. Thus, the typing constraint is expressed by the Arrow In Constraint. As the *to* object of an inherited arrow is always subclass of the *to* object of the original arrow, every program in our data model satisfies the Subtyping Constraint. Thus, no checking of the Subtyping Constraint is needed. However, our data model enforces the Arrow Isa Constraint which says that if an arrow a is subclass of an arrow a' then $from(a)$ must be subclass of $from(a')$ and $to(a)$ must be subclass of $to(a')$. Though it is not required that the Arrow Isa Constraint be enforced by a data model, we feel that it imposes a discipline that protects against the declaration of erroneous information.

Many object-oriented data models that define their semantics based on logic, such as [29, 8, 6, 31, 18], do not consider inheritance of typing properties[6]. Yet, in many applications, reasoning on the structural definitions of the data is a necessity [32]. A property inherited by a class provides information about the class that may not be obvious by just browsing

[6]These data models consider non-monotonic inheritance.

through the specialization hierarchy. In our data model, the arrows inherited by a class and their Isa and $Risa$ relations give useful information about the class semantics (many examples supporting this claim are given in [3]).

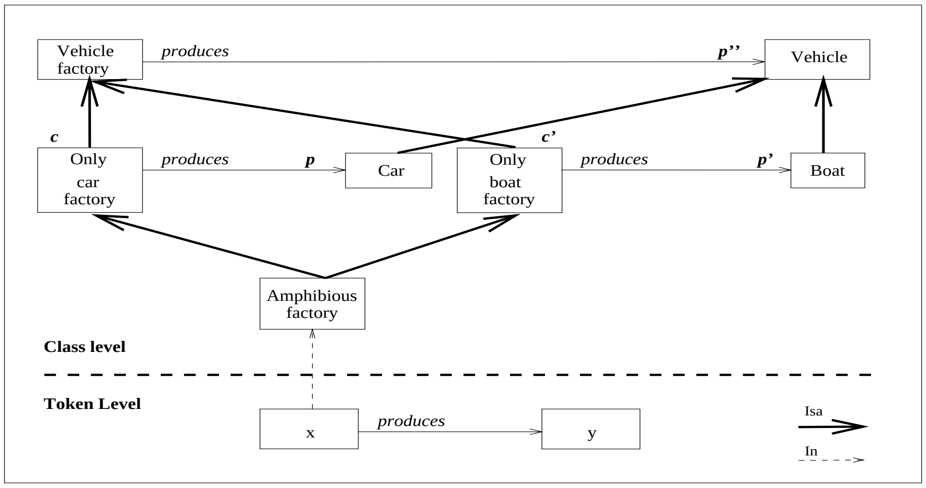

Figure 7. Example of typing properties having same label and same semantics

We will compare our approach to typing property inheritance and typechecking with that of F-logic [26], terminological languages [7, 33, 28, 4], DOT [41, 42], QUIXOTE [45], and LOGIN [1]. First, we present common limitations of these data models:

(i) These data models support $Risa$ only implicitly, based on property labels. Specifically, if two classes c, c' have each a property with the same label, then it is assumed that the two properties have same semantics (for common subclasses of c, c'). In our data model, this is translated to either (a) the two properties are related through $Risa$, in the case that c, c' are related through Isa, or (b) the two properties are related through $Risa$ to a common, more generic property, in the case that c, c' are not related through Isa. For example, in Figure 7, it is assumed that the properties p, p' have same semantics. In our data model, this information could have been expressed explicitly by declaring the relations $Risa(p, p'')$ and $Risa(p', p'')$.

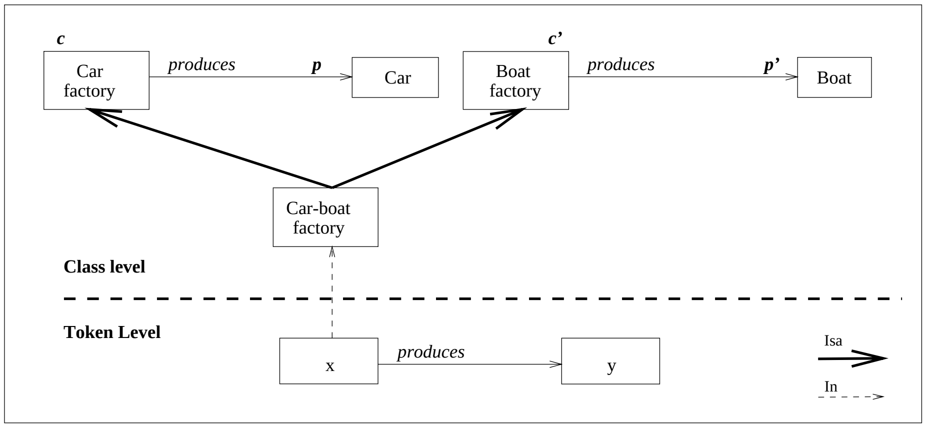

Figure 8. Example of typing properties having same label but not same semantics

Clearly, the approach followed by the other data models may not always give the desirable results. For an example, refer to Figure 8, where the properties p, p' have the same label *produces*. In our data model, the properties p, p' are not considered to have same semantics (for common subclasses of c, c'), as they are not related through *Risa* to a common, more generic property. Thus, the class *Car-boat-factory* inherits two different properties: one from p, and another from p'. 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, the other data models will derive that the value of the inherited property *produces* of *Car-boat-factory* is subclass of both *Car* and *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 x with a property *produces* whose value is y. The other data models can only express the case where y is both car and boat. This case can also be expressed in our data model by declaring that the property *produces* of x is instance of both properties p and p'. However, the case where y *is not* a boat cannot be expressed in the other data models, whereas it can be expressed in our data model (by declaring that the property *produces* of x is instance of the property p, only).

Additionally, in our data model, derived *Risa* relations (through the *Risa* Rules) may relate properties with different labels. Obviously, this is not possible in the other data models.

(ii) The other data models do not support *Isa* relations between properties[7]. Additionally, the *Isa* and *Risa* relations between properties do not interact to refine the value of the inherited property. For example, in Figure 3, the other data models will not derive that the value of the property *collects* inherited by *Rich_painting_collector* from *Painting_collector* is subclass of *Expensive_art_object*. Thus, our data model provides a finer value for the inherited property. Additionally, in our data model, derived *Isa* and *Risa* relations between the inherited property and other properties provide useful information.

(iii) In the other data models, the value of an inherited property is not a *known* class[8]. This implies that, in contrast to our data model, no reasoning can be done based on the "specific" value of the inherited property. For an example, refer to Figure 7. In the other data models, the value of the inherited property *produces* of *Amphibious factory* is an *unknown* class (with superclasses *Car* and *Boat*). In contrast, in our data model, the value of the inherited property *produces* of *Amphibious factory* is $meet(Car, Boat)$, which is a *known* class.

We now present the approaches followed by F-logic, terminological languages, DOT, and QUIXOTE, with respect to typing property inheritance and typechecking.

F-LOGIC [26]

F-logic is a powerful deductive object-oriented language that supports inheritance of typing properties. Let c be a class having a typing property with label l. Then, the statement $c[l \Rightarrow \{d_1, ..., d_n\}]$ asserts that if an instance of c has a property with label l then its value is instance of each class $d_1, ..., d_n$. Typing property inheritance is supported by the *F-logic rules*:

(i) If $c'[l \Rightarrow D']$ and $Isa(c, c')$ then $c[l \Rightarrow D']$.
(ii) If $c[l \Rightarrow D]$, $c[l \Rightarrow D']$ then $c[l \Rightarrow D \cup D']$.

The fact that an object o has a non-typing property with label l and the value of the property is d, is asserted by the statement $o[l \rightarrow d]$.

[7] We should mention that the *Isa* relation between properties is supported by TELOS [30], NIAM [43], and OSAM* [38]. However, these models do not formalize typing property inheritance based on logic.

[8] We say that a class is *known* if it is an explicit class or the meet of explicit classes.

For example, in F-logic, the information in Figure 7, is expressed as follows:
Isa relations between classes, are declared as shown in the figure,
$Vehicle\_factory[produces \Rightarrow \{Vehicle\}]$, $Only\_car\_factory[produces \Rightarrow \{Car\}]$,
$Only\_boat\_factory[produces \Rightarrow \{Boat\}]$, $In(x, Amphibious\_factory)$,
$x[produces- > y]$.

From the F-logic rules, it is derived that: $Amphibious\_factory[produces \Rightarrow \{Car, Boat\}]$. This indicates that if an instance of the class $Amphibious\_factory$ has a property *produces* then the value of the property should be an instance of both Car and $Boat$. F-logic enforces the Typing constraint. Thus, in Figure 7, if object y is not an instance of both Car and $Boat$ then the corresponding program is considered to be invalid.

Due to F-logic rule (ii), every F-logic program satisfies the Subtyping Constraint and no checking of the constraint is needed. No analog to Arrow Isa Constraint exists in F-logic. For example, in Figure 7, assume that Car is not subclass of $Vehicle$. In contrast to our data model, this will not cause any constraint violation in F-logic, and $Only\_car\_factory[produces \Rightarrow \{Car, Vehicle\}]$ will be derived.

Terminological Languages

Data models based on *terminological languages*[9], such as [7, 33, 28, 4], support the taxonomic representation of *concepts* (correspond to individual simple-classes, in our data model), and typing property inheritance. Concepts are described in terms of other concepts and necessary and sufficient conditions on their properties (called *roles*), through a set of operators. Concepts are put into a hierarchy where a concept is below the concepts it specializes. Concepts inherit properties and property value constraints from the concepts above in the hierarchy. Local and inherited property value constraints are combined to refine the value of the property. In fact, our comments on F-logic with respect to typing property inheritance, Typing and Subtyping constraints, apply also to terminological languages.

In contrast to our data model, terminological languages do not treat concepts and their properties in a uniform way. Specifically, they do not consider properties to be concepts, on their own. Therefore, properties are not organized in an inheritance hierarchy, and do not have properties. Additionally, terminological languages do not support meta-concepts. Therefore, they cannot support uniform querying at instance and schema level. We consider this to be a severe limitation, as adding meaning to the data, should be accompanied by convenient ways of querying the schema (through a meta-schema).

DOT [41, 42]

The knowledge representation model DOT describes property values using the Isa relation and supports typing property inheritance. In this model, the In relation is not distinguished from Isa (for this reason, instead of the term *subclass*, we will use the term *specialization*). For example, in Figure 7, the In relation should be replaced by Isa. The value of a property with label l of an object c is denoted by $c.l$. Property inheritance is supported by the *DOT rule*: If $Isa(c, c')$ then $Isa(c.l,\ c'.l)$.

In DOT, the information in Figure 7, is expressed as follows:
Isa relations between classes, are declared as shown in the figure,
$Isa(Vehicle\_factory.produces,\ Vehicle)$, $Isa(Only\_car\_factory.produces,\ Car)$,
$Isa(Only\_boat\_factory.produces,\ Boat)$, $Isa(x,\ Amphibious\_factory)$,
$Isa(y,\ x.produces)$.

[9]Terminological languages are sometimes refer to as *description logics*.

From the DOT rule, it follows that $Amphibious\_factory$ inherits the property $produces$, and the object $Amphibious\_factory.produces$ is a specialization of Car, $Boat$, $Vehicle$, $Only\_car\_factory.produces$, $Only\_boat\_factory.produces$, and $Vehicle\_factory.produces$. Additionally, it follows that the object $x.produces$ is a specialization of $Amphibious\_factory.produces$. From this, it follows (due to Isa transitivity) that y is a specialization of both Car and $Boat$. However, it will not be derived that *Amphibious_factory.produces = meet(Car,Boat)*, as in our data model.

We would like to mention an important difference between DOT and our data model. In DOT, in the case that the relations $Isa(y, Car)$ and $Isa(y, Boat)$ have not been explicitly declared, they are derived from the DOT rule. Therefore, no checking of the Typing Constraint is needed. In contrast, in our data model, in F-logic, and in terminological languages, if $In(y, Car)$ and $In(y, Boat)$ have not been explicitly declared then the Typing Constraint will be violated. Additionally, due to DOT rule, every DOT program satisfies the Subtyping constraint. Therefore, similarly to F-logic and to terminological languages, no checking of the Subtyping Constraint is needed.

QUIXOTE [45]

QUIXOTE is a deductive object-oriented language that supports typing property inheritance. As in DOT, the In relation is not distinguished from Isa. Property inheritance is supported by the QUIXOTE rules:
(i) if $Isa(c, c')$ then $Isa(c.l,\ c'.l)$, and
(ii) if $Isa(c.l,\ d)$ and $Isa(c.l,\ d')$ then $Isa(c.l,\ meet(d, d'))$.
Rule (i) refines the value of the inherited properties, and rule (ii) refines them further. For an example, in Figure 7, it will be derived that $Isa(Amphibious\_factory.produces,$ $meet(Car, Boat))$. However, it will not be derived that $Amphibious\_factory.produces =$ $meet(Car, Boat)$, as in our data model. Similarly to DOT, every QUIXOTE program satisfies the Typing and Subtyping Constraints. Thus, no typechecking is needed.

LOGIN [1]

LOGIN is a logic programming language that supports specialization hierarchies and typing property inheritance through a unification algorithm. Similarly to DOT and QUIXOTE, LOGIN does not distinguishes between the In and Isa relations. Additionally, similarly to QUIXOTE, LOGIN assumes that the set of explicit objects with the Isa relation forms a lattice.

If an object c has a property p with value d then this is expressed with the statement (called ψ-*term*[10]) $c(p \Rightarrow d)$. Let c' be an object. If c' has a property p then the unification of the ψ-terms $c(p \Rightarrow d)$ and $c'(p \Rightarrow d')$ gives the ψ-term $meet(\{c, c'\})(p \Rightarrow$ $meet(\{d, d'\}))$. Otherwise, the unification of the ψ-terms $c(p \Rightarrow d)$ and $c'()$ gives the ψ-term $meet(\{c, c'\})(p \Rightarrow d)$. For example, in LOGIN, the information in Figure 7, is expressed as follows:
The Isa relations between classes, as shown in the figure,
$Vehicle\_factory(produces \Rightarrow Vehicle)$, $Only\_car\_factory(produces \Rightarrow Car)$,
$Amphibious\_factory(produces \Rightarrow Boat)$, $Isa(x,\ Amphibious\_factory)$,
$x(produces \Rightarrow y)$.

Note that the meet of $Only\_car\_factory$ and $Only\_boat\_factory$ is $Amphibious\_factory$

[10] ψ-terms are in general much more expressive, as they support nested sub-ψ-terms and *coreference* constraints.

and the meet of Car and $Boat$ is $Amphibious$. Therefore, the unification of $Amphibious\_factory$ with $Only\_car\_factory$ and $Only\_boat\_factory$ gives $Amphibious\_factory(produces \Rightarrow Amphibious)$.

LOGIN checks the Typing Constraint, as not all programs satisfy it. For example, in Figure 7, if y is not declared as a specialization of both Car and $Boat$ the the program will be considered to be invalid. Note that, in LOGIN, the analogous to both the Arrow In Constraint and Arrow Isa Constraint is the Tuping Constraint. This is because, LOGIN does not differentiate between the In and Isa relations. For example, in Figure 7, if Car is not declared as a specialization of $Vehicle$ then the Typing Constraint will be violated.

8. Conclusions

In this paper, we elaborated on the *semantics* of an enhanced object-oriented semantic network, where multiple instantiation, multiple specialization, and meta-classes are supported for both entities and properties.

The user defines objects and relations between objects through *declarations*. A set of declarations that satisfies certain syntactical conditions makes up a *program*.

Reasoning in our data model is done through a number of (built-in) *inference rules* that allow for derivations both at instance and schema level. In addition to the inference rules, a number of (built-in) *system constraints* exist for checking the validity of the program. Through the inference rules, new objects are derived, as well as new In, Isa, and $Risa$ relations between both explicit and derived objects. In particular, these rules relate inherited properties to other properties through Isa and $Risa$ relations. Such relations not only give useful information about the inherited properties but also refine the value of these properties.

Specifically, each program P has a set of *models* that satisfy the inference rules and the declarations in P. We defined a partial ordering between the models of P and proved that P has a least model, say M. If M satisfies the system constraints then we consider it as the *semantics* of P.

In this paper, properties are inherited from classes to subclasses. However, property inheritance can also take place from a class to its instances. This kind of inheritance is called *instance inheritance*. For example, assume that class *Art collector* has a property *collects* with value *Art object* and let p denote this property. Every instance o of *Art collector* can instance-inherit a property from p indicating that o collects art objects. A possible value of the instance-inherited property is *Art object*. However, this value can be refined based on relations of p with other properties. Other information declared by the user, may be utilized for this purpose as well. We currently investigate new relations allowing the user to express information about the class of the values of properties of o. In particular, the new relations should be able to express that (i) all properties of o which are instances of a property p have value in a class d, and (ii) there is a property of o which is instance of p and has value in d. This will allow the representation of incomplete knowledge, as the specific values of the properties of o may be unknown.

APPENDIX

Here, we give the proofs of all Propositions and Theorems.

Proposition 6.1 Let P be a program. The relation "$\leq$" is a partial ordering over the models of P is a partial order (up to model equivalence).

Proof:

Obviously, $\leq$ is reflexive and transitive. We will now prove that $\leq$ is antisymmetric (up to model equivalence). Let M, M' be two models of P such that $M \leq M'$ and $M' \leq M$. We will prove that $M \equiv M'$.

In the proof, we will use the usual symbols to denote components of M and the same symbols with a prime to denote the corresponding components of M'.

As $M \leq M'$, there is a mapping $\mathcal{F} : O \rightarrow O'$ that satisfies the conditions of Definition 6.9. Additionally, as $M' \leq M$, there is a mapping $\mathcal{F}' : O' \rightarrow O$ that satisfies the conditions of Definition 6.9. First, we will show that F' is the inverse of F.

Let $o \in O$ and $o' = \mathcal{F}(o)$. We will show that $\mathcal{F}'(o') = o$. Assume that $\mathcal{F}'(o') = o_1$. As $M \leq M'$, it holds that $ref(o) \subseteq ref'(o')$. As $M' \leq M$, it holds that $ref'(o') \subseteq ref(o_1)$. Thus, $ref(o) \subseteq ref(o_1)$. As $ref(o) \neq \{\}$, there is $r \in O_{ref}$ such that $obj(r) = o$ and $obj(r) = o_1$. As obj is a function, it follows that $o = o_1$.

Let $o' \in O'$ and $o = \mathcal{F}'(o')$. We can similarly show that $\mathcal{F}(o) = o'$. Thus, F' is the inverse of F. From this, it follows that F is a bijective mapping.

From condition 4 of Definition 6.9 and Domain Rule 3, it follows that: o is in I, A, H, T, and C iff $\mathcal{F}(o)$ is in I',A', H', T', and C', respectively.

Let $o \in E$ and let $o' = \mathcal{F}(o)$. As $name$ is a total function on E, $name(o)$ is defined. It follows from condition 1 of Definition 6.9 that $name'(o')$ is defined. Thus, $o' \in E'$.

Let $a \in EA$ and let $a' = \mathcal{F}(a)$. From the above, it follows that $a' \in EA'$. As $label$ is total function on EA, $label(a)$ is defined. Similarly, $label'(a')$ is defined. From Name Rule 2 and the fact that $name(a) = name'(a')$, it follows that $label(a) = label'(a')$. Thus, $\forall\, a \in EA, \quad label(a) = label'(\mathcal{F}(a'))$.

From condition 2 of Definition 6.9 and the fact that $\mathcal{F} = \mathcal{F}'^{-1}$ it follows that $\forall\, o \in O, \quad ref(o) = ref'(\mathcal{F}(o'))$.

From condition 1 of Definition 6.9 and the fact that obj is function, it follows that $\forall\, s \in \mathcal{L}$, $\mathcal{F}(meet(s)) = meet'(\mathcal{F}(s))$.

We will now prove the following statements:

1. $\forall\, c, a' \in O$, if $e\_inh(c,a')$ is defined then $\mathcal{F}(e\_inh(c,a')) = e\_inh'(\mathcal{F}(c), \mathcal{F}(a'))$ and $\mathcal{F}(to(e\_inh(c,a'))) = to'(e\_inh'(\mathcal{F}(c), \mathcal{F}(a')))$
2. $\forall\, c, a' \in O$, if $a\_inh(c,a')$ is defined then $\mathcal{F}(a\_inh(c,a')) = a\_inh'(\mathcal{F}(c), \mathcal{F}(a'))$.

Let $EM_0 = E \cup meet(\mathcal{L})$. Recall that $einh(c,a')$ is defined iff $Isa(c, from(a'))$. From this and condition 4 of Definition 6.9, it follows that $\forall\, c \in EM_0,\ a' \in EA$, $e\_inh(c,a')$ is defined iff $e\_inh'(\mathcal{F}(c), \mathcal{F}(a'))$ is defined. Similarly, $\forall\, c \in EM_0,\ a' \in EA, \quad a\_inh(c,a')$ is defined iff $a\_inh'(\mathcal{F}(c), \mathcal{F}(a'))$ is defined.

From condition 2 of Definition 6.9, the Reference Rules, and the fact that obj is a function, it follows that (i) $\forall\, c \in EM_0,\ a' \in EA$, if $e\_inh(c,a')$ is defined then $\mathcal{F}(e\_inh(c,a')) = e\_inh'(\mathcal{F}(c), \mathcal{F}(a'))$ and $\mathcal{F}(to(e\_inh(c,a'))) = to'(e\_inh'(\mathcal{F}(c), \mathcal{F}(a')))$, and (iii) $\forall\, c \in EM_0,\ a' \in EA$, if $a\_inh(c,a')$ is defined then $\mathcal{F}(a\_inh(c,a')) = a\_inh'(\mathcal{F}(c), \mathcal{F}(a'))$.

Obviously, the previous statements now hold if we replace EM_0 by $EM_1 = e\_inh(EM_0, EA) \cup a\_inh(EM_0, EA)$. We can continue like that recursively. Thus, we have proved statements 1, 2.

From condition 3 of Definition 6.9, it follows that $\forall\ c, a' \in O$, if $cand\_cl(c, a')$ is defined then $\mathcal{F}(cand\_cl(c, a')) = cand\_cl'(\mathcal{F}(c), \mathcal{F}(a'))$. From this and Approximate Inheritance Rule 4, it follows that $\mathcal{F}(to(a\_inh(c, a'))) = to'(a\_inh'(\mathcal{F}(c), \mathcal{F}(a')))$.

From the above, it now easily follows that $\forall\ a \in A$, $\mathcal{F}(from(a)) = from'(\mathcal{F}(a))$ and $\mathcal{F}(to(a)) = to'(\mathcal{F}(a))$.

From condition 4 of Definition 6.9 and the fact that $\mathcal{F} = \mathcal{F}^{-1}$ it follows that $\forall\ o, o' \in O$, $Rel \in \{In, Isa, Risa\}$, $Rel(o, o')$ holds iff $Rel'(\mathcal{F}(o), \mathcal{F}(o'))$ holds.

It now follows that $M \equiv M'$. $\diamond$ $\square$

Theorem 6.1 Every program P has a least model.

Proof:
We will prove the theorem in two steps.
Step 1: We will construct a structure M and show that M is a model of P.

First, according to $indiv()$, $hybrid()$, and $arrow()$ declarations of P, we construct objects, insert them in E, I, H, A, and name them. Then, we relate these objects according to the $isa()$, $risa()$, and $in()$ declarations of P.

We execute all the inference rules except Approximate Inheritance Rule 4, until the fixpoint (rules with $\leftrightarrow$ are executed in both directions). Call the result F. Then, we execute Approximate Inheritance Rule 4. This will assign a meet class to the *to* object of inherited arrows. Then, we execute again all the inference rules except Approximate Inheritance Rule 4 until fixpoint. Call the result M. To show that M satisfies all the inference rules, it is enough to show that $\forall\ c, a' \in O$, $cand(c, a')$ is the same in F and M.

Assume that there exist c, a' such that $cand(c, a')$ is different in F and M. Thus, there is an explicit arrow a, such that $to(a)$ is in $cand(c, a')$ with respect to M but not with respect to F. Thus, there a relation $Isa(e\_inh(c, a'), a)$ that is not derived in F and is derived in M using Exact Inheritance Rule 3. This implies that there is a relation $Risa(a\_inh(c', a''), a_0)$, where $c', a'', a_0 \in O$, which is not derived in F and is derived in M using the Risa Rules 5 and 6. This relation is needed in order to derive $Isa(e\_inh(c, a'), a)$. However, if this is the case then $Risa(e\_inh(c', a''), a_0)$ should have been derived in F using Exact Inheritance Rule 3 and Risa Rule 6[11]. From $Risa(e\_inh(c', a''), a_0)$, the relation $Isa(e\_inh(c, a'), a)$ would have been derived in F, in the same way $Isa(e\_inh(c', a'), a)$ is derived from $Risa(a\_inh(c', a''), a_0)$ in M. However, this is impossible because we have assumed that $Isa(e\_inh(c, a'), a)$ is not derived in F.

Thus, M satisfies all inference rules. Due to conditions 1 and 2 of Definition 6.4 (Declaration program), *name* is a function. Additionally, all other constraints of a structure are satisfied. Thus, M is a model.

Step 2: We will now show that M is the least model of P.

Let M' be any model[12] of P. We will show that $M \leq M'$.

[11]Note that Exact Inheritance Rule 3 is a special case of Risa Rule 5 (arrow a_1 in Risa Rule 5 is replaced by $e\_inh(c, a')$ and the condition on the *to* objects is eliminated).

[12]Symbols of structure components with a prime, denote components of M'.

It is easy to see from the construction of M that for each $o \in O$, there is an object $o' \in O'$ with the same reference. We consider the mapping $\mathcal{F} : O \rightarrow O'$ that maps every object $o \in O$ to an object $o' \in O'$ such that $ref(o) \cap ref'(o') \neq \emptyset$. From the fact that obj is a function, there is only one such o'. Obviously, $\mathcal{F}$ satisfies conditions 1 and 2 of Definition 6.9.

From the fact that $\forall\ o \in O,\ \ name(o) = name'(\mathcal{F}(o))$ and the fact that obj is function, it follows that $\forall\ s \in \mathcal{L}$, $\mathcal{F}(meet(s)) = meet'(\mathcal{F}(s))$.

We will now show the following statements:

1. $\forall\ c, a' \in O$, if $e\_inh(c,a')$ is defined then $\mathcal{F}(e\_inh(c,a')) = e\_inh(\mathcal{F}(c), \mathcal{F}(a'))$ and $\mathcal{F}(to(e\_inh(c,a'))) = to'(e\_inh'(\mathcal{F}(c), \mathcal{F}(a')))$
2. $\forall\ c, a' \in O$, if $a\_inh(c,a')$ is defined then $\mathcal{F}(a\_inh(c,a')) = a\_inh(\mathcal{F}(c), \mathcal{F}(a'))$.

Let $EM_0 = E \cup meet(\mathcal{L})$. From the construction of M, if $Isa(o,o')$ holds, for $o, o' \in EM_0$ then $Isa'(\mathcal{F}(o), \mathcal{F}(o'))$ holds. From this and the fact that $einh(c,a')$ is defined iff $Isa(c, from(a'))$. it follows that $\forall\ c \in EM_0,\ a' \in EA$, if $e\_inh(c,a')$ is defined then $e\_inh'(\mathcal{F}(c), \mathcal{F}(a'))$ is defined. Similarly, $\forall\ c \in EM_0,\ a' \in EA$, if $a\_inh(c,a')$ is defined then $a\_inh'(\mathcal{F}(c), \mathcal{F}(a'))$ is defined.

From the Reference Rules, and the fact that obj is a function, it follows that (i) $\forall\ c \in EM_0,\ a' \in EA$, if $e\_inh(c,a')$ is defined then $\mathcal{F}(e\_inh(c,a')) = e\_inh'(\mathcal{F}(c), \mathcal{F}(a'))$ and $\mathcal{F}(to(e\_inh(c,a'))) = to'(e\_inh'(\mathcal{F}(c), \mathcal{F}(a')))$, and (iii) $\forall\ c \in EM_0,\ a' \in EA$, if $a\_inh(c,a')$ is defined then $\mathcal{F}(a\_inh(c,a')) = a\_inh'(\mathcal{F}(c), \mathcal{F}(a'))$.

Obviously, the previous statements now hold if we replace EM_0 by $EM_1 = e\_inh(EM_0, EA) \cup a\_inh(EM_0, EA)$. We can continue like that recursively. Thus, we have proved statements 1, 2.

From the above and the construction of M, it follows that $\mathcal{F}$ also satisfies conditions 3 and 4 of Definition 6.9. Thus, Theorem 6.1 follows. $\diamond$ $\square$

Proposition 6.2 Let P be a valid program. Then, for each $o \in E$, it holds that either $o \in I$, or $o \in A$, or $o \in H$.

Proof:
As M is the least model of P, each object in E has been created through a declaration $indiv()$, $arrow()$, or $hybrid()$. Therefore, $E \subseteq I \cup A \cup H$. Statement 1 now follows directly from the fact that the sets I, A, and H are disjoint. $\square$

Proposition 6.3 Let P be a valid program. Let a' be an arrow class and let c be a subclass of $from(a')$. Then, the arrow $e\_inh(c,a')$ is restriction subclass of $a\_inh(c,a')$.

Proof:
From Exact Inheritance Rule 2, it follows that $Risa(e\_inh(c,a'), a')$. From Approximate Inheritance Rule 2, it follows that $Risa(a\_inh(c,a'), a')$. Therefore, it follows from Exact Inheritance Rule 3, $Isa(e\_inh(c,a'), a\_inh(c,a'))$. As $Risa(a\_inh(c,a'), a')$, it follows from Risa Rule 6 that $Risa(e\_inh(c,a'), a\_inh(c,a'))$. $\diamond$ $\square$

Proposition 6.4 Let P be a valid program. Let a_0, a_1 be arrow classes and c be a class. If $Risa(a_0, a_1)$ and $Isa(c, from(a_0))$ then $e\_inh(c,a_0) = e\_inh(c,a_1)$ and $a\_inh(c,a_0) = a\_inh(c,a_1)$.

Proof:
We will first prove that $e\_inh(c, a_0) = e\_inh(c, a_1)$. As $Risa(e\_inh(c, a_1),\ a_1)$ (from Exact Inheritance Rule 2) and $Risa(a_0, a_1)$ (from hypothesis), it follows from Exact Inheritance Rule 3 that $Isa(e\_inh(c, a_1), a_0)$. As $Risa(e\_inh(c, a_0),\ a_0)$, it follows from Exact Inheritance Rule 3 that $Isa(e\_inh(c, a_1),\ e\_inh(c, a_0))$.

As $Isa(e\_inh(c, a_1), a_0)$, $Risa(a_0, a_1)$, and $Risa(e\_inh(c, a_1),\ a_1)$, it follows from Risa Rule 6 that $Risa(e\_inh(c, a_1), a_0)$. As $Isa(e\_inh(c, a_0),\ a_0)$, it follows from Exact Inheritance Rule 3, that $Isa(e\_inh(c, a_0),\ e\_inh(c, a_1))$. Therefore from Exact Inheritance Rule 7, it follows that $e\_inh(c, a_0) = e\_inh(c, a_1)$.

We will now prove that $a\_inh(c, a_0) = a\_inh(c, a_1)$. As $e\_inh(c, a_0) = e\_inh(c, a_1)$, it follows that $cand\_cl(c, a_0) = cand\_cl(c, a_1)$. Thus, $to(a\_inh(c, a_0)) = to(a\_inh(c, a_1))$. Additionally, from Exact Inheritance Rule 5, it follows that $to(a_0) \in cand\_cl(c, a_1)$. Thus, it holds that $Isa(to(a\_inh(c, a_1)), to(a_0))$. From Approximate Inheritance Rule 2, it holds that $Risa(a\_inh(c, a_1),\ a_1)$. As $Risa(a_0, a_1)$, it now follows from Risa Rule 5 that $Isa(a\_inh(c, a_1), a_0)$. As $Risa(a\_inh(c, a_0),\ a_0)$, it now follows from Risa Rule 5, that $Isa(a\_inh(c, a_1),\ a\_inh(c, a_0))$. Similarly, it follows that $Isa(a\_inh(c, a_0),\ a\_inh(c, a_1))$. Therefore from Approximate Inheritance Rule 3, it follows that $a\_inh(c, a_0) = a\_inh(c, a_1)$.
$\diamond$ $\square$

Proposition 6.5 Let P be a valid program. Let a' be an explicit class and let c be a subclass of $from(a')$. If the arrow $a\_inh(c, a')$ is restriction subclass of an explicit arrow a then $e\_inh(c, a') = e\_inh(c, a)$ and $a\_inh(c, a') = a\_inh(c, a)$.

Proof:
We will first prove that $e\_inh(c, a') = e\_inh(c, a)$.

As $Risa(e\_inh(c, a'), a\_inh(c, a'))$ (Proposition 6.3) and $Risa(a\_inh(c, a'), a)$ (from hypothesis), it follows that $Risa(e\_inh(c, a'),\ a)$. From Exact Inheritance Rule 2, it holds that $Risa(e\_inh(c, a),\ a)$. Thus, from Exact Inheritance Rule 3, it follows that $Isa(e\_inh(c, a'), e\_inh(c, a))$. Similarly, from Exact Inheritance Rule 3, it follows that $Isa(e\_inh(c, a), e\_inh(c, a'))$. Therefore, it follows from Exact Inheritance Rule 7 that $e\_inh(c, a') = e\_inh(c, a)$.

We will now prove that $a\_inh(c, a') = a\_inh(c, a)$. As $e\_inh(c, a') = e\_inh(c, a)$, it follows that $cand\_cl(c, a') = cand\_cl(c, a)$. Therefore, $to(a\_inh(c, a')) = to(a\_inh(c, a))$. From hypothesis, it follows that $Risa(a\_inh(c, a'),\ a)$. From Approximate Inheritance Rule 2, it holds that $Risa(a\_inh(c, a),\ a)$. Thus, from Risa Rule 5, it follows that $Isa(a\_inh(c, a'),\ a\_inh(c, a))$. Again, from Risa Rule 5, it follows that $Isa(a\_inh(c, a),\ a\_inh(c, a'))$. Therefore, it follows from Approximate Inheritance Rule 3 that $a\_inh(c, a') = a\_inh(c, a)$. $\diamond$ $\square$

Proposition 6.6 Let P be a valid program. There is no explicit class or meet class d such that d is subclass of the to object of an exact inherited arrow.

Proof:
Assume that there is such a class d and exact inherited arrow a. To derive that $Isa(d, to(a))$, the Exact Inheritance Rule 4 should have been used. This implies that the d should be the to object of an exact inherited arrow $e\_inh(c, a')$. However, this is impossible because d is an explicit class or meet class. $\diamond$ $\square$

References

[1] Ait-Kaci, R. Nasr, LOGIN: A Logic Programming Language with Built-in Inheritance, *Journal of Logic Programming*, 3(3), 187-215, (1986).

[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, P. Constantopoulos, N. Spyratos, Restriction by Specialization and Schema Derivations, Technical Report TR176, Institute of Computer Science, Foundation for Research and Technology-Hellas, October, (1996). Available from ftp.ics.forth.gr, tech-reports/1996/ 1996.TR176.Specialization_Restriction_Schema_Derivations.ps.gz.

[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] E. Bertino, D. Montesi, Towards a Logical-Object Oriented Programming Language for Databases, *Proc. of the Third Intern. Conference for Databases*, pp. 168-183, (1992).

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

[8] 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*, pp.411-430, (1991).

[9] G.H.W.M. Bronts, S.J. Brouwer, C.L.J. Martens, H.A. Proper, A Unifying Object Role Modelling Theory, *Information Systems*, 20(3), 213-235, (1995).

[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] A. Cardelli, P. Wegner, On Understanding Types, Data Abstraction, and Polymorphism, *ACM Computing Surveys*, 1(4), 471-522, (1985).

[12] , P.P. Chen, The Entity-Relationship Model: Toward a Unified View of Data, *ACM Transactions on Database Systems*, 1(1), pp. 9-36, (1976).

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

[14] 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/.

[15] 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).

[16] 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).

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

[18] G. Dobbie, R. Topor, Resolving Ambiguities caused by Multiple Inheritance, *Proc. of the Fourth Intern. Conference on Deductive and Object-Oriented Databases*, pp. 265- 280, (1995).

[19] G. Engels, et al., Conceptual Modeling of Database Applications using an Extended ER Model, *Data & Knowledge Engineering*, 9(4), pp. 157-204, (1992).

[20] Associative Networks, New York: Academic Press, N. Findler (ed.), (1979).

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

[22] A.H.M. ter Hofstede, Th.P. van der Weide, Expressiveness in Conceptual Data Modelling, *Data & Knowledge Engineering*, 10(1), 65-100, (1993).

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

[24] 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).

[25] M.H. Jamil, L.V.S. Lakshmanan, ORLOG: A Logic for Semantic Object-Oriented Models, *Proc. of the First Intern. Conference on Information and Knowledge Management*, pp. 584-592, (1992).

[26] 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).

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

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

[29] K. Lee, S. Lee, An Object-Oriented Approach to Data/Knowledge Modeling Based on Logic, *Proc. of the 1990 IEEE Sixth Intern. Conference*, pp.289-294, (1990).

[30] 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).

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

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

[33] 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).

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

[35] L. Rowe, M. Stonebraker, The POSTGRES Data Model, *Proc. of the Intern. Conference on Very Large Data Bases*, 83-96, (1987).

[36] Principles of Semantic Networks: Explorations in the Representation of Knowledge, J.F. Sowa (ed.), Morgan Kaufmann, (1991).

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

[38] 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).

[39] T.J. Teorey, D. Yang, J.P. Fry, A Logical Design Methodology for Relational Databases Using the Extended Entity-Relationship Model, *ACM Computing Surveys*, 18(2), 197-222, (1986).

[40] M. Theodorakis, P. Constantopoulos, Context-Based Naming in Information Bases, International Journal of Cooperative Information Systems, 6(3&4), 269-292, (1997).

[41] 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*, pp. 391-410, (1991).

[42] 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*, pp. 246-264, (1995).

[43] 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.), pp.537-590, (1982).

[44] W.A. Woods, What's in a Link: Foundation for Semantic Networks, *Representation and Understanding: Studies in Cognitive Science*, D. Bobrow, A. Collins (eds.), pp. 35-82, (1975).

[45] 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*, pp. 89-99, (1993).