On the Definition of Semantic Network Semantics

Anastasia Analyti\textsuperscript{1}, Nicolas Spyrotos\textsuperscript{3,*}, Panos Constantopoulos\textsuperscript{1,2}

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

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 \textit{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.

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 [38, 18, 32] 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 [10, 19, 34, 37, 5, 21, 30, 27, 17, 20, 7] 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) [12, 13, 14]. 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, instantiation and specialization are expressed through the In and Isa relations, respectively. However, as we demonstrated in [2], an additional form of property specialization is needed that represents property value refinement. This led us to introduce the restriction isa (Risa) relation.

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 [2], 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 semantics of semantic networks supporting multiple instantiation, multiple specialization, and metaclasses. Moreover, the present paper provides a formal account for several semantic constructs introduced in an earlier paper [2].

The rest of the paper is organized as follows: Section 2 describes our view of the real world. Section 3 presents constructs for representing the real world, and defines object derivation. 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 The “real world”

In this section, we present our view of the real world. First, we describe real world objects in terms of intension and extension. Based on this description, we relate real world objects through the *instantiation* and *specialization* relations\(^1\). Finally, we define inheritance in the real world.

Real world objects, or *real objects* for short, are distinguished with respect to their nature into: *real individuals*, *real arrows*, and *real hybrids*.

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

- **Real arrows**: These are concrete or abstract properties/binary relationships\(^2\) of real objects. More precisely, a real arrow represents a property or a relationship of a real

---

\(^1\) Throughout the paper, we use the term *relation* with its mathematical meaning.

\(^2\) We do not make the distinction between property and binary relationship, as our approach is common to both.
object $O$ with value a real object $O'$. The real objects $O$, $O'$ are called the from and to object of the real arrow, respectively. The from object of a real arrow $A$ is denoted by $\text{from}(A)$ and the to object by $\text{to}(A)$.

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

- **Real hybrids**: These are abstractions that refer collectively to real 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 real arrows whose to object is a real individual or a real hybrid. Yet, the from object of a real arrow can be of any type. In particular, it can be another real arrow.

Real objects are also distinguished with respect to their concreteness, into real tokens and real classes.

- **Real Tokens**: These are concrete real objects, such as the individual my car, and the arrow produced from opel to my car.

- **Real Classes**: These are abstract real objects, in the sense that they refer collectively to a set of real objects that are considered similar in some respect. Real classes whose instances are classes are called real meta-classes.

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

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

Our view of real world objects as tokens or classes on one hand, and as individuals, arrows, or hybrids on the other, follows quite closely the structural part of the knowledge representation language Telos [27, 22].

We assume that a real object is defined by a set of constraints, called the intension of the object. For a real class, the intension determines the set of real objects to which the real class refers collectively. We call this set the extension of the real class. The extension
of a real individual class is a set of real individuals. The extension of a real arrow class from a real class $O$ to a real class $O'$ is a set of real arrows from objects in the extension of $O$ to objects in the extension of $O'$. The extension of a real hybrid is a set of real individuals, arrows, or hybrids.

Real objects can be related to real classes through the *instance of* relation.

**Definition 2.1 IN relation**

If a real object $O$ belongs to the extension of a real class $C$ then we say that $O$ is an *instance of* $C$, and we denote it by $\text{IN}(O, C)$. A real object can be an instance of zero, one, or more classes (multiple instantiation). ♦

![Diagram](image)

**Figure 1:** Example of relations between real objects

For an example, refer to Figure 1(a), where the real arrow token *collects from* $\text{ART COLLECTOR X}$ to $\text{ART OBJECT Y}$ is instance of the real arrow class *collects from* $\text{ART COLLECTOR}$ to $\text{ART OBJECT}$.

### 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 real classes and expresses inclusion of class extensions. The RISA relation relates pairs of real arrow classes and expresses property value refinement.

**Definition 2.2 ISA relation**

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

- **Case 1:** $o'$ is a subclass of $o$.
- **Case 2:** $o'$ is a superclass of $o$.
- **Case 3:** $o'$ and $o$ are disjoint.
Case 1: \( o \) and \( o' \) are real individual classes.

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

Case 2: \( o \) and \( o' \) are real arrow classes.

We say that \( o \) is \textit{subclass} of \( o' \), denoted by \( \text{ISA}(o, o') \), if it holds that:

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

Case 3: \( o \) is a real class and \( o' \) is a real hybrid.

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

In all other cases, \( \text{ISA} \) is undefined.

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

See for example Figure 1(b), where the real class \texttt{painting collector} refers to art collectors that collect paintings (but may also collect other art objects). The real arrow class \texttt{collects from painting collector to painting} is subclass of the real arrow class \texttt{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 real arrow classes. We say that \( A \) is a \textit{restriction subclass} of \( A' \), denoted by \( \text{RISA}(A, A') \), if the following hold:

(i) \( \text{ISA}(A, A') \), and

(ii) for any \( X \), if \( \text{IN}(X, A') \) and \( \text{IN}(\text{from}(X), \text{from}(A)) \) then \( \text{IN}(X, A) \). \( \Diamond \)

For example, see Figure 1(b), where the real class \texttt{only painting collector} refers to art collectors that collect only paintings. The real arrow \texttt{collects of only painting collector} is a restriction subclass of the real arrow \texttt{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.
2.2 Real Arrow Inheritance

Let $A'$ be a real arrow class from a real class $C'$ to a real class $D'$. Let $C$ be a subclass of $C'$. Our purpose is to define the real arrow inherited by $C$ from $A'$, denoted by $inh(C, A')$.

Recall that the real arrow $A'$ expresses a property of $C'$ with values in $D'$ that refers collectively to a set, say $E'$, of real arrows. The \textit{from} objects of arrows in $E'$ are instances of $C'$ and their \textit{to} objects are instances of $D'$. The question is what part of $A'$ expresses a property of $C$. It is this part that we shall call the real arrow inherited by $C$ from $A'$. Obviously, this part is the set $E$ of real arrows in $E'$ whose \textit{from} object is an instance of $C$. Having determined the extension $E$ of the inherited real arrow, it remains to determine its \textit{to} object. Let $D(C, A')$ be the real object that refers collectively to the \textit{to} objects of the real arrows in $E$. We take $D(C, A')$ to be the \textit{to} object of $inh(C, A')$.

By this reasoning, we define the real arrow inherited by $C$ from $A'$, as follows:

\textbf{Definition 2.4 Inherited real arrow}

Let $A'$ be a real arrow class from a real class $C'$ to a real class $D'$ and let $C$ be a subclass of $C'$. Let $D(C, A')$ be the class whose extension is the set of \textit{to} objects of all instances $X$ of $A'$ such that \textit{from}(X) is an instance of $C$.

We define $inh(C, A')$ to be the real arrow class from $C$ to $D(C, A')$ whose extension consists of all instances of $A'$ whose \textit{from} object is an instance of $C$. The real arrow $inh(C, A')$ is called the real arrow inherited by $C$ from $A'$.

![Diagram of real arrow inheritance](image)

Figure 2: Example of real arrow inheritance

For an example, refer to Figure 2. Note that $D(C, A')$ is subclass of \textit{painting}. This is
because, as \( C \) refers to art collectors that collect only paintings, the to objects of all real arrows in \( E \) are instances of painting.

In [2], we prove several properties of inherited real arrows. In particular, we prove that \( inh(C, A') \) is a restriction subclass of \( A' \).

3 The Model

In the previous section, we described real world objects. However, only a fragment of the real world is mapped in the information base. This fragment is delimited by the needs of the user and by his imperfect knowledge of the real world. The latter implies that not all objects of interest and their relations are represented in the information base. We refer to the representation of the real world in the information base, as the model.

A real object is represented in the model by what we call a model object, or simply an object. An object in the model is called individual, arrow, or hybrid, if it represents a real individual, a real arrow, or a real hybrid, respectively. An object is called token or class if it represents a real token or a real class, respectively.

Objects are also characterized as explicit or derived. Explicit objects are declared by the user or the system, whereas derived objects are derived by the system (see subsection 3.1). Additionally, objects are distinguished in user objects and system objects. The system objects are built-in objects with controlled access by users.

Object relations

Two objects in the model may be related through the relations, In, Isa, and Risa, that represent the relations IN, ISA, and RISA in the real world, respectively. Relations between explicit objects are either declared (by the user or the system) or derived by the system. In contrast, relations between derived objects and other objects are derived by the system, only.

If \( In(o, c) \) holds then we say that \( o \) is an instance of \( c \), or \( c \) is a class of \( o \). The set of instances of a class \( c \) is called the extension of \( c \). If \( Isa(c, c') \) holds then we say that \( c \) is a subclass of \( c' \), or that \( c' \) is a superclass of \( c \). If \( Risa(a, a') \) holds then we say that \( a \) is a restriction subclass of \( a' \), or that \( a' \) is a restriction superclass of \( a \).

Object names and object references

Let \( IND, HYB, \) and \( ARR \) be sets of symbols. Each explicit individual and hybrid has a
name in \(IND\) and \(HYB\), respectively. Two different individuals and hybrids cannot have
the same name. Each explicit arrow has a label in \(ARR\) which is unique within its \textit{from}
object. In other words, two arrows with the same \textit{from} object must have different labels.
However, two arrows with different \textit{from} objects can have the same label. The name of an
explicit arrow \(a\) is formed by the name of \(\text{from}(a)\) and the label of \(a\). Specifically, if \(f\)
is the name of \(\text{from}(a)\) and \(l\) is the label of \(a\) then the name of \(a\) is \(f/l\). For example, if an
arrow \(a\) has label \textit{produced} and the name of its \textit{from} object is \textit{Car-company} then the name
of \(a\) is \textit{Car-company/produced}.

Each object has an associated set of references. The name of an explicit object \(o\) is
always a reference to \(o\). However, as we shall see in Section 3.1, \(o\) may have references other
than its name. In that same section, we will see how the references of derived objects are
formed. In fact, the need to introduce references is due mainly to the presence of derived
objects. Intuitively, a reference is formed from (explicitly defined) names.

3.1 Derived Objects

In our data model, there are four kinds of derived objects: \textit{meet classes}, \textit{exact inherited}
arrow, \textit{to objects of exact inherited arrows}, and \textit{approximate inherited arrows}.

Intuitively, \textit{meet classes} are intersections of explicit individual and/or hybrid classes,
whereas, exact and approximate inherited arrows are properties inherited by subclasses.
In particular, the \textit{to} object of an approximate inherited arrow is a \textit{meet class}. The exact
inherited arrows are auxiliary derived objects that are introduced for the derivation of
approximate inherited arrows.

3.1.1 Meet Classes

Intuitively, the \textit{meet class} of a set of classes is the intersection of the classes in the set.
However, as classes in the model represent classes of the real world, the \textit{meet class} must be
defined with respect to the real world.

Consider the poset of real individual and hybrid classes \(R\) with the partial order ISA.
Let \(S\) be a subset of \(R\). The greatest lower bound of \(S\), denoted by \text{meet}(S), is the real
class whose extension is the intersection of the extensions of the real classes in \(S\).

\textbf{Definition 3.1 Meet class}

Let \(s\) be a set of explicit individual or hybrid classes and let \(S\) be the set of real classes

10
represented by the classes in $s$. We define the *meet class* of $s$, denoted by $meet(s)$, to be the class that represents the real class $meet(S)$. 

We distinguish two cases, depending on whether $s$ has a least class 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 $meet(s) = c$. This is because the real class represented by $c$ coincides with $meet(S)$.

*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.

A reference to $meet(s)$ is derived from the names of the objects in $s$. Specifically, we define $\text{meet}(\{\text{name}(o) \mid o \in s\})$ to be a reference to $meet(s)$, where the function $\text{name}$ maps an object to its name.

For an example, refer to Figure 4. The set of classes $s = \{d', d_0, d_1\}$ has no least class, so $\text{meet}(\{d', d_0, d_1\})$ is a derived object. This derived object is subclass of $d'$, $d_0$, and $d_1$. Note that $\text{Painting}, \text{Expensive.art.object}, \text{Art.object}$ are the names of the objects $d'$, $d_0$, and $d_1$, respectively. Therefore, a reference to $\text{meet}(\{d', d_0, d_1\})$ is $\text{meet}(\{\text{Painting}, \text{Expensive.art.object}, \text{Art.object}\})$.

### 3.1.2 Inherited Arrows

Let $a'$ be an explicit arrow class from a class $c'$ to a class $d'$ and let $c$ be a subclass of $c'$. Let $A'$, $C$ be the real objects represented by $a'$, $c$, respectively. Recall that $\text{inh}(C, A')$ denotes the real arrow inherited by $C$ from $A'$. This real arrow has from object $C$ and to object $D(C, A')$. We next define the *exact arrow inherited* by $c$ from $a'$ with respect to the real world. This concept is needed for the definition of *inherited arrows*, later on.

**Definition 3.2 Exact inherited arrow**

Let $a'$ be an explicit arrow class from a class $c'$ to a class $d'$ and let $c$ be a subclass of $c'$. Let
$A'$, $C$ be the real objects represented by $a'$, $c$, respectively. We call exact arrow inherited by $c$ from $a'$, denoted by $e_{\text{inh}}(c, a')$, the arrow that represents the real arrow $\text{inh}(C, A')$.

Every exact inherited arrow $e_{\text{inh}}(c, a')$ is a derived object. Obviously, the from object of $e_{\text{inh}}(c, a')$ is $c$. The to object of $e_{\text{inh}}(c, a')$ is a derived class, denoted by $d(c, a')$, that represents $D(C, A')$.

A reference to $e_{\text{inh}}(c, a')$ is derived 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 $e_{\text{inh}}(r, n)$ is a reference to $e_{\text{inh}}(c, a')$.

Figure 3: (a) Arrow inheritance in the real world, (b) Arrow inheritance in the model

As $\text{inh}(C, A')$ is restriction subclass of $A'$, $e_{\text{inh}}(c, a')$ is restriction subclass of $a'$. For example, in Figure 3, the exact inherited arrow $e_{\text{inh}}(c, a')$ represents the inherited real arrow $\text{inh}(C, A')$. Note that, $\text{inh}(C, A')$ is restriction subclass of $A'$ and $e_{\text{inh}}(c, a')$ is restriction subclass of $a'$.

We would like to emphasize that we do not know what exactly the class $d(c, a')$ is. What we do know are $\text{Isa}$ and $\text{Risa}$ relations of $a'$ with other arrows, as well as $\text{Isa}$ relations of $c$ with other classes (as declared by the user or the system). Based on these relations and using inference rules, we can derive $\text{Isa}$ relations of $\text{inh}(c, a')$ to other arrows. If we derive that $\text{inh}(c, a')$ is subclass of an arrow $a$, then we can derive that $d(c, a')$ is subclass of $\text{to}(a)$. 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 $\text{cand.}\text{d}(c, a')$ and is computed in subsection 4.3. Intuitively, this set provides an “approximation” of the
class $a(c, a')$.

Roughly speaking, as the classes in $\text{cand}_{\mathcal{L}}(c, a')$ are explicit, their “intersection” is known. Therefore, we would like to define an inherited arrow that “approximates” $\text{e.inh}(c, a')$ and its to object corresponds to the “intersection” of the classes in $\text{cand}_{\mathcal{L}}(c, a')$. We denote this arrow by $a_{\text{inh}}(c, a')$ and call it approximate inherited arrow or inherited arrow, for short. Let $S$ be the set of real classes represented by the classes in $\text{cand}_{\mathcal{L}}(c, a')$. Obviously, $a_{\text{inh}}(c, a')$ should represent a real arrow from $C$ to $\text{meet}(S)$ which is restriction subclass of $A'$. In [2], we prove that there is a unique real arrow with these properties. Thus, $a_{\text{inh}}(C, A')$ is well defined.

**Definition 3.3 Inherited arrow**

Let $a'$ be an explicit arrow class from a class $c'$ to a class $d'$ and let $c$ be a subclass of $c'$. Let $A', C$ be the real objects represented by $a', c$, respectively. We define the approximate arrow inherited by $c$ from $a'$, or arrow inherited by $c$ from $a'$, for short, the arrow that represents the real arrow $a_{\text{inh}}(C, A')$. The arrow inherited by $c$ from $a'$ is denoted by $a_{\text{inh}}(c, a')$.

Every inherited arrow $a_{\text{inh}}(c, a')$ is a derived object. Obviously, the from object of $a_{\text{inh}}(c, a')$ is $c$ and the to object of $a_{\text{inh}}(c, a')$ is the class $\text{meet}(\text{cand}_{\mathcal{L}}(c, a'))$ (that represents $\text{meet}(S)$).

A reference to $a_{\text{inh}}(c, a')$ is derived 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 $a_{\text{inh}}(r, n)$ is a reference to $a_{\text{inh}}(c, a')$.

**Examples**

In Figure 3(b), we have $\text{cand}_{\mathcal{L}}(c, a') = \{\text{Art object}\}$. Therefore, as Art object is the least object of $\text{cand}_{\mathcal{L}}(c, a')$, it follows that the to object of $a_{\text{inh}}(c, a')$ is Art object. Note that $a_{\text{inh}}(c, a')$ represents the real arrow $a_{\text{inh}}(C, A')$ which is restriction subclass of $A'$ and goes from $C$ to Art object.

For another example, refer to Figure 4, where we have that $\text{cand}_{\mathcal{L}}(c, a_0) = \{d', d_0, d_1\}$. The to object of $a_{\text{inh}}(c, a')$ is the derived class $\text{meet}(\text{cand}_{\mathcal{L}}(c, a'))$.

An extended discussion and illustrative examples regarding inherited arrows can be found in [2].
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. We say that an inference rule r represents a rule R if R results from r after replacing the objects and relations in r with their counterparts in the real world. For example, the Isa Rule 2 (given below) represents the rule:

For all real classes $C_1$, $C_2$, $C_3$,  $\text{ISA}(C_1, C_2) \land \text{ISA}(C_2, C_3) \Rightarrow \text{ISA}(C_1, C_3)$.

All inference rules in our model are sound, as they represent rules holding in the real world.

$O$: set of objects  \hspace{1cm} T: set of tokens  \hspace{1cm} AC: set of arrow classes  \hspace{1cm} O_{sy} : set of system objects
$I$: set of individuals  \hspace{1cm} C: set of classes  \hspace{1cm} EA: set of explicit arrows
$H$: set of hybrids  \hspace{1cm} E: set of explicit objects  \hspace{1cm} EAC: set of explicit arrow classes
$A$: set of arrows  \hspace{1cm} IC: set of individual classes
$\text{L}$: powerset of explicit individual and hybrid classes

Figure 5: 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 $L = \mathcal{P}((IC \cup H) \cap E)$ to denote the powerset of explicit individual and hybrid classes. Additionally, we use $O_{sys}$ to denote the set of system objects. These notations are shown in Figure 5.

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.

<table>
<thead>
<tr>
<th>ISA RULES</th>
</tr>
</thead>
<tbody>
<tr>
<td>Rule 1: $\forall c \in C, ; Isa(c,c)$</td>
</tr>
<tr>
<td>Rule 2: $\forall c_1, c_2, c_3 \in C, ; Isa(c_1, c_2) \land Isa(c_2, c_3) \Rightarrow Isa(c_1, c_3)$</td>
</tr>
<tr>
<td>Rule 3: $\forall o \in O, c, c' \in C, ; In(o,c) \land Isa(c,c') \Rightarrow In(o,c')$</td>
</tr>
</tbody>
</table>

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.

<table>
<thead>
<tr>
<th>RISA RULES</th>
</tr>
</thead>
<tbody>
<tr>
<td>Rule 1: $\forall a, a' \in AC, ; Risa(a,a') \Rightarrow Isa(a,a')$</td>
</tr>
<tr>
<td>Rule 2: $\forall x \in A, a_1, a_2 \in AC, ; Risa(a_1, a_2) \land In(x, a_2) \land In(from(x), from(a_1)) \Rightarrow In(x, a_1)$</td>
</tr>
<tr>
<td>Rule 3: $\forall a \in AC, ; Risa(a,a)$</td>
</tr>
<tr>
<td>Rule 4: $\forall a_1, a_2, a_3 \in AC, ; Risa(a_1, a_2) \land Risa(a_2, a_3) \Rightarrow Risa(a_1, a_3)$</td>
</tr>
<tr>
<td>Rule 5: $\forall a_1, a_2, a_3 \in AC,$</td>
</tr>
<tr>
<td>$Isa(a_1, a_3) \land Risa(a_2, a_3) \land Isa(from(a_1), from(a_2)) \land Isa(to(a_1), to(a_2)) \Rightarrow Isa(a_1, a_2)$</td>
</tr>
<tr>
<td>Rule 6: $\forall a_1, a_2, a_3 \in AC, ; Isa(a_1, a_2) \land Isa(a_2, a_3) \land Risa(a_1, a_3) \Rightarrow Risa(a_1, a_2)$</td>
</tr>
</tbody>
</table>

15
Risa Rules 1 and 2 reflect the definition of RISA. We refer to [2], for illustrative examples and discussions regarding the Risa Rules.

4.2 Meet Rules

In section 3.1.1, we defined the meet classes with respect to the real world. Recall that $\mathcal{L}$ is the powerset of explicit individual and hybrid classes and $\text{meet} : \mathcal{L} \to C$. The Meet Rules derive meet classes and relate them to other objects.

**MEET RULES**

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

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

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

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

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

Note that Meet Rules 2 and 3 derive Isa relations for $\text{meet}(s)$, where $s \in \mathcal{L}$. From Meet Rule 1, it follows that if $c$ is the least class of $s$ then $\text{meet}(s)$ is the explicit class $c$. Additionally, it will follow from the definition of the program semantics that the inverse is also true. Specifically, if $\text{meet}(s)$ is an explicit class $c$ then $c$ is the least class of $s$. Therefore, it follows that if $\text{meet}(s)$ 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 $\text{meet}(s) = \text{meet}(s - \{o'\})$, as expected. This result is an easy consequence of Meet Rules 2,3,5 and Isa Rule 2. For example, in Figure 4, it holds that $\text{meet}(d', d_0, d_1) = \text{meet}(d', d_0)$. 

16
### 4.3 Inheritance Rules

In section 3.1.2, we defined the exact inherited arrows and (approximate) inherited arrows with respect to the real world. 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 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:** \( \forall a' \in EAC, \ c \in C, \)

\[
\begin{align*}
\text{Isa}(c, \text{from}(a')) & \iff \exists e_{\text{inh}}(c, a') \in AC \\
\exists e_{\text{inh}}(c, a') \in AC & \Rightarrow \text{from}(e_{\text{inh}}(c, a')) = c
\end{align*}
\]

**Rule 2:** \( \forall a' \in EAC, \ c \in C, \ Risa(e_{\text{inh}}(c, a'), a') \)

**Rule 3:** \( \forall a' \in EAC, \ c \in C, \ a_0, a_1 \in AC, \)

\[
\text{Isa}(e_{\text{inh}}(c, a'), a_1) \wedge Risa(a_0, a_1) \wedge \text{Isa}(c, \text{from}(a_0)) \Rightarrow \text{Isa}(e_{\text{inh}}(c, a'), a_0)
\]

**Rule 4:** \( \forall a' \in EAC, \ c \in C, \ a \in AC, \)

\[
\text{Isa}(e_{\text{inh}}(c, a'), a) \Rightarrow \text{Isa}(\text{to}(e_{\text{inh}}(c, a')), \text{to}(a))
\]

**Rule 5:** \( \forall a' \in EAC, \ d \in C \cap E, \ c \in C, \)

\[
\text{Isa}(\text{to}(e_{\text{inh}}(c, a'))), d \Rightarrow d \in \text{cand}_{\text{eil}}(c, a')
\]

**Rule 6:** \( \forall a' \in EAC, \ c \in C, \ x \in A, \)

\[
\text{In}(x, e_{\text{inh}}(c, a')) \Rightarrow \text{In}(\text{to}(x), \text{to}(e_{\text{inh}}(c, a')))
\]

**Rule 7:** \( \forall a', a'' \in EAC, \ c \in C, \)

\[
\text{Isa}(e_{\text{inh}}(c, a'), e_{\text{inh}}(c, a'')) \wedge \text{Isa}(e_{\text{inh}}(c, a''), e_{\text{inh}}(c, a')) \Rightarrow e_{\text{inh}}(c, a') = e_{\text{inh}}(c, a'')
\]

---

Exact Inheritance Rule 4, refines the class `to(e_{inh}(c, a'))`. That is, it relates the `to` object of \( e_{\text{inh}}(c, a') \) 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., \( \text{cand}_{\text{eil}}(c, a') \)). As this rule is the only rule that defines \( \text{cand}_{\text{eil}}(c, a') \), the set \( \text{cand}_{\text{eil}}(c, a') \) includes all superclasses of `to(e_{inh}(c, a'))`.

For an example of Exact Inheritance Rule 3, refer to Figure 4. First, observe that arrow \( e_{\text{inh}}(c, a') \) is a restriction subclass of `a'` (Exact Inheritance Rule 2). Additionally, observe
that \( a' \) is subclass of \( a_1 \). Thus, from \( Isa \) Rule 2 (transitivity), we derive that \( \text{\_inh}(c, a') \) is subclass of \( a_1 \). On the other hand, arrow \( a_0 \) is restriction subclass of \( a_1 \). As \( c \) is subclass of \( \text{from}(a_0) \), we have all three conditions of Exact Inheritance Rule 3 satisfied. Thus, we obtain that \( \text{\_inh}(c, a') \) is subclass of \( a_0 \).

For an example of Exact Inheritance Rule 4, we continue with our previous example (Figure 4). As \( \text{\_inh}(c, a') \) is subclass of \( a' \), the \textit{to} object of \( \text{\_inh}(c, a') \) (denoted by \( d(c, a') \)) is subclass of \( d' \). As \( \text{\_inh}(c, a') \) is subclass of \( a_1 \), \( d(c, a') \) is subclass of \( d_1 \). Additionally, as \( \text{\_inh}(c, a') \) 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 \( \text{cand.\_cl}(c, a') = \{d', d_0, d_1\} \). Additionally, it follows from Meet Rule 3 that \( d(c, a') \) is subclass of \( \text{meet}(\{d', d_0, d_1\}) \).

The \textit{Approximate Inheritance Rules} derive inherited arrows and relate them to other arrows.

### APPROXIMATE INHERITANCE RULES

**Rule 1:** \( \forall a' \in EAC, \ c \in C, \)

\[
\text{Isa}(c, \text{from}(a')) \Leftrightarrow \exists \text{\_inh}(c, a') \in AC
\]

\[
\exists \text{\_inh}(c, a') \in AC \Rightarrow \text{from}(\text{\_inh}(c, a')) = c
\]

**Rule 2:** \( \forall a' \in EAC, \ c \in C, \) \( \text{Risa}(a, \text{\_inh}(c, a'), a') \)

**Rule 3:** \( \forall a', a'' \in EAC, \ c \in C, \)

\[
\text{Isa}(a, \text{\_inh}(c, a'), a, \text{\_inh}(c, a'')) \land \text{Isa}(a, \text{\_inh}(c, a''), a, \text{\_inh}(c, a')) \Rightarrow a, \text{\_inh}(c, a') = a, \text{\_inh}(c, a'')
\]

**Rule 4:** \( \forall a' \in EAC, \ c \in C, \) \( \exists a, \text{\_inh}(c, a') \in AC \Rightarrow \text{to}(a, \text{\_inh}(c, a')) = \text{meet}(\text{cand.\_cl}(c, a')) \)

For example, in Figure 4, \( \text{\_inh}(c, a') \) is restriction subclass of \( a' \) and the \textit{to} object of \( a, \text{\_inh}(c, a') \) is \( \text{meet}(\{d', d_0, d_1\}) \).

**5 System Constraints**

The following \textit{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 *Isa Antisymmetry Constraint* (below) is justified by the ISA antisymmetry property. The inference rule corresponding to this constraint 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*: The Isa relation must relate only classes.
   \[ \text{Isa} \subseteq C \times C \]

2. *Isa Antisymmetry Constraint*: The Isa relation must be antisymmetric.
   \[ \forall a, b \in O, \quad \text{Isa}(a, b) \land \text{Isa}(b, a) \Rightarrow a = b \]

3. *Arrow Isa Constraint*: 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(\( a' \)).
   \[ \forall a, a' \in A, \quad \text{Isa}(a, a') \Rightarrow \text{Isa}(\text{from}(a), \text{from}(a')) \land \text{Isa}(\text{to}(a), \text{to}(a')) \]

4. *System Object Isa Constraint*: System objects must be subclasses only of system objects.
   \[ \forall o \in O^{sys}, \ c \in O, \quad \text{Isa}(o, c) \Rightarrow c \in O^{sys} \]

**IN CONSTRAINTS**

1. *In Domain Constraint*: An object can be instance of a class, only.
   \[ \text{In} \subseteq O \times C \]

2. *Arrow In Constraint*: If an arrow \( x \) is instance of an arrow \( a \) then from(\( x \)) must be instance of from(\( a \)) and to(\( x \)) must be instance of to(\( a \)).
   \[ \forall x, a \in A, \quad \text{In}(x, a) \Rightarrow \text{In}(\text{from}(x), \text{from}(a)) \land \text{In}(\text{to}(x), \text{to}(a)) \]

3. *System Object In Constraint*: System objects must be instances of system objects, only.
   \[ \forall o \in O^{sys}, \ c \in O, \quad \text{In}(o, c) \Rightarrow c \in O^{sys} \]

**CONCRETENESS CONSTRAINTS**

1. *Concreteness Constraint*: No token can be a class, and vice versa.
   \[ T \cap C = \emptyset \]

19
The Arrow In and Isa Constraints are used for typechecking (as we shall see in section 7).

Intuitively, the System Object In Constraint expresses that the user cannot declare system objects as instances of user objects. Similarly, the System Object Isa Constraint expresses that the user cannot declare system objects as subclasses of user objects. This is because system objects should not inherit properties from user objects. However, system objects can have instances and subclasses which are user objects.

6 Formal Account of the Model

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).

Objects are distinguished into user objects and system objects. User objects are declared by the user, whereas system objects are built-in objects. The system objects include, in particular, the data types supported by the system, as well as the data values in the domains of these types. For example, the data types can be atomic data types (such as, Int, Float, Boolean, Char) plus structured data types (such as interval, collection, or cross product). We denote by $DT$ the set of data types supported by the system and by $DV$ the set of all values in the domains of the data types. That is, $DV = \cup \{\text{domain}(dt) \mid dt \in DT\}$, where $\text{domain}(dt)$ is the domain of the data type $dt$. For example, $\text{dom}(\text{Int})$ can be the set of C++ integers.

During their interaction, the user and the system use symbols in order to name the objects of interest and their relationships. Examples of such symbols are the data types and the data values just mentioned. However, other symbols are needed as well. The set
of all symbols that the user and the system are allowed to use for naming purposes is what we call the *alphabet*.

**Definition 6.1 Alphabet**

We call *alphabet* a set of symbols\(^3\) that consists of the following mutually disjoint subsets.

\(IND\): The set of symbols for naming individuals. This set includes the reserved symbols *Indiv* and *IndivClass*, as well as all symbols in the sets \(DT\) and \(DV\), mentioned above.

\(ARR\): The set of symbols for naming arrows. This set includes the reserved symbols *Arrow* and *ArrowClass*.

\(HYB\): The set of symbols for naming hybrids. This set includes the reserved symbols *Object*, *Token*, *Class*, and *Hybrid*.  

In all definitions of this section, we assume a fixed alphabet.

In order to declare objects of interest, the user or the system uses names built from symbols in the alphabet and the auxiliary symbol “/” as follows:

**Definition 6.2 Object names**

We define the arrow names \(A_{nam}\) and object names \(O_{nam}\), as follows:

\[
A_{nam} = \{ n/l_1/\ldots/l_k \mid n \in IND \cup HYB, \ l_1, \ldots, l_k \in ARR \} \\
O_{nam} = IND \cup HYB \cup A_{nam}
\]

With names, the user can refer only to explicit objects. In order to refer to both explicit and derived objects the user needs references. Arrow references \(A_{ref}\) and object references \(O_{ref}\) are defined in steps as follows:

**Definition 6.3 Object references**

We first define the step \(i\) arrow references \(A^i_{ref}\) and object references \(O^i_{ref}\), as follows:

for \(i = 0\):

\[
A^0_{ref} = A_{nam}, \quad O^0_{ref} = O_{nam} \cup \text{meet}(P(IND \cup HYB))
\]

\(^3\)No symbol contains “/”,“(“ or “)”. 

21
for $i > 0$:

$$A_{ref}^i = \{ e_{\omega}^{inh}(r, n) \mid r \in O_{ref}^{i-1}, \ n \in \text{nam} \} \cup \{ a_{\omega}^{inh}(r, n) \mid r \in O_{ref}^{i-1}, \ n \in \text{nam} \}$$

$$O_{ref}^i = A_{ref}^i \cup \{ d(r, n) \mid r \in O_{ref}^{i-1}, \ n \in \text{nam} \}$$

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

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

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 $e_{\omega}^{inh}(c, a')$, where the reference to $c$ was constructed at step $i - 1$.

---

**Figure 6: System declarations**

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

**Definition 6.4 Declarations**

A *declaration* is one of the following expressions:

- $\text{indiv}(n)$, where $n \in IND$.  

---
It declares an individual with name \( n \).

- \( hybrid(n) \), where \( n \in HYB \).
  It declares a hybrid with name \( n \).
- \( arrow(f, l, t) \), where \( f \in O_{\text{nam}}, \ l \in ARR, \) and \( t \in IND \cup HYB \).
  It declares an arrow with label \( l \) that goes from an object with name \( f \) to an individual or hybrid with name \( t \).
- \( in(n, n') \), where \( (n, n') \in (IND \times IND) \cup (A_{\text{nam}} \times A_{\text{nam}}) \cup (O_{\text{nam}} \times HYB) \).
  It declares an object with name \( n \) to be instance of an object with name \( n' \).
- \( isa(n, n') \), where \( (n, n') \in (IND \times IND) \cup (A_{\text{nam}} \times A_{\text{nam}}) \cup (O_{\text{nam}} \times HYB) \).
  It declares an object with name \( n \) to be subclass of an object with name \( n' \).
- \( isa(n, n') \), where \( n, n' \in A_{\text{nam}} \).
  It declares an arrow with name \( n \) to be restriction subclass of an arrow with name \( n' \).

\( \diamond \)

Intuitively, the type of the \( in \) declaration indicates that individuals can have only individual instances, arrows can have only arrow instances, whereas, hybrids can have instances of any kind. Analogous comment can be made for the \( isa \) declaration.

Declarations are also made by the system in order to declare objects of generic interest, as well as relations between these objects. We shall refer to these declarations as \textit{system declarations}. Figure 6(a) shows the system declarations declaring the system objects. Figure 6(b) shows, pictorially, the system declarations declaring relations between system objects.

\textbf{Definition 6.5 Declaration program}

We call \textit{declaration program} or \textit{simply program}, any set of declarations together with the system 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 \( n \in IND \cup HYB \) then there is a declaration \( indiv(n) \) or \( hybrid(n) \),
   - if \( n = n'/l \), where \( l \in ARR \), then there is a declaration \( arrow(n', l, t) \).
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.

When talking about a program, the system declarations will always be (tacitly) assumed as part of it. Moreover, we assume that reserved symbols (i.e., \( \text{Indiv} \), \( \text{IndivClass} \), and so on) are not used by the user for naming individuals, hybrids, or arrows.

### 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 6.6 Semantic structure**

A semantic structure is defined by a tuple: \( S = < O, \ell, \text{DerObj}, \text{Ref} > \), where: \( \ell = < \text{In}, \text{Isa}, \text{Risa} > \), \( \text{DerObj} = < \text{meet}, \text{e.inh}, \text{a.inh} > \), and \( \text{Ref} = < \text{name}, \text{label}, \text{obj} > \), 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. \( O \) must satisfy the Domain Rules (see further on).

We shall use the following notations: \( IC = I \cap C \), \( AC = A \cap C \), \( EA = E \cap A \), and \( EAC = E \cap AC \) denote the sets of individual classes, arrow classes, explicit arrows, and explicit arrow classes of \( S \), respectively. Additionally, \( L = \mathcal{P}(IC \cup H) \cap E \) denotes the powerset of explicit individual and hybrid classes. These notations are shown in Figure 5.

- \( \ell : A \to O \) and \( \text{to} : A \to I \cup H \) are total functions that associate each arrow with its from and to objects.

- \( \text{In} \subseteq (I \times I) \cup (A \times A) \cup (O \times H) \) is a relation expressing the instance relation between objects.
• \(I\sa \subseteq (I \times I) \cup (A \times A) \cup (O \times H)\) is a relation expressing the subclass relation between objects. \(I\sa\) must satisfy the \(I\sa\) Rules (see subsection 4.1).

• \(R\sa \subseteq A \times A\) is a relation expressing the restriction subclass relationship between arrows. \(R\sa\) must satisfy the \(R\sa\) Rules (see subsection 4.1).

• \(\text{meet} : \mathcal{L} \rightarrow C\) is a total function that associates a set \(s \in \mathcal{L}\) with the class \(\text{meet}(s)\). The function \(\text{meet}\) must satisfy the \(\text{Meet}\) Rules (see subsection 4.2).

• \(e\cdot \text{inh} : C \times EAC \rightarrow AC\) 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 \(e\cdot \text{inh}\) must satisfy the \(\text{Exact Inheritance Rules}\) (see subsection 4.3).

• \(a\cdot \text{inh} : C \times EAC \rightarrow AC\) 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 \(a\cdot \text{inh}\) must satisfy the \(\text{Approximate Inheritance Rules}\) (see subsection 4.3).

• \(\text{name} : E \rightarrow O_{\text{name}}\) is a total, one-to-one function that associates an explicit object with its name. The function \(\text{name}\) must satisfy the \(\text{Name Rules}\) (see further on).

• \(\text{label} : EA \rightarrow A\text{RR}\) is a total function that associates an explicit arrow with its label.

• \(\text{obj} : O_{\text{ref}} \rightarrow O\) is a partial, surjective function that associates object references to objects. The function \(\text{obj}\) must satisfy the \(\text{Reference Rules}\) (see further on).

---

**DOMAIN RULES**

**Rule 1:** The set \(O\) must include the system objects \(O_{\text{sys}}\). The set \(O_{\text{sys}}\) consists of three sets of objects, namely \(I^{\text{sys}}\), \(H^{\text{sys}}\), and \(A^{\text{sys}}\), defined as follows:

\[
I^{\text{sys}} = \{\#\text{Indiv}, \#\text{IndivClass}\} \cup \{\#p \mid p \in PT \cup PV\},
\]

\[
H^{\text{sys}} = \{\#\text{Hybrid}, \#\text{Object}, \#\text{Token}, \#\text{Class}\}
\]

\[
A^{\text{sys}} = \{\#\text{Arrow}, \#\text{ArrowClass}\}
\]

\[
O_{\text{sys}} = I^{\text{sys}} \cup H^{\text{sys}} \cup A^{\text{sys}} \subseteq O
\]

**Rule 2:** \(I^{\text{sys}} \subseteq I\), \(A^{\text{sys}} \subseteq A\), \(H^{\text{sys}} \subseteq H\).

**Rule 3:** \(\forall i \in O, \; \text{In}(i, \#\text{Indiv}) \Leftrightarrow i \in I\)

\(\forall h \in O, \; \text{In}(h, \#\text{Hybrid}) \Leftrightarrow h \in H\)

25
∀ a ∈ O, \( \text{In}(a, \# \text{Arrow}) \iff a \in A \)
∀ o ∈ O, \( \text{In}(o, \# \text{Token}) \iff o \in T \)
∀ c ∈ O, \( \text{In}(c, \# \text{Class}) \iff c \in C \)

**Rule 4:** ∀ i ∈ I, \( \text{In}(i, \# \text{Indiv}) \land \text{In}(i, \# \text{Class}) \implies \text{In}(i, \# \text{IndivClass}) \).

**Rule 5:** ∀ a ∈ A, \( \text{In}(a, \# \text{Arrow}) \land \text{In}(a, \# \text{Class}) \implies \text{In}(a, \# \text{ArrowClass}) \).

**Rule 6:** ∀ i ∈ I, \( \text{In}(i, \# \text{IndivClass}) \implies \text{Isa}(i, \# \text{Indiv}) \)
∀ a ∈ A, \( \text{In}(a, \# \text{ArrowClass}) \implies \text{Isa}(a, \# \text{Arrow}) \)

**Rule 7:** ∀ c ∈ C, \( \text{Isa}(c, \# \text{Object}) \)

---

### NAME RULES

**Rule 1:** Naming of system objects is as follows:
\[ \forall \# n \in P^{90} \cup H^{90}, \quad \text{name(\# n)} = n, \quad \text{e.g., name(\# Object)} = \text{Object}. \]
\[ \forall \# l \in A^{90}, \quad \text{label(\# l)} = l \]

**Rule 2:** Let \( a \) be an explicit arrow. If \( f \) is the name of from(\( a \)) and \( l \) is the label of \( a \) then the name of \( a \) is \( fl \).
\[ \forall a \in EA, \quad f \in O_{nam}, \quad l \in ARR, \quad \text{name(from}(a)) = f \land \text{label}(a) = l \implies \text{name}(a) = fl \]

---

### REFERENCE RULES

**Rule 1:** The name of any explicit object is a reference to the object.
\[ \forall o \in E, \quad n \in O_{nam}, \quad \text{name}(o) = n \implies \text{obj}(n) = o \]

**Rule 2:** If \( s \) is a set of explicit individual and hybrid classes then \( \text{meet(\{\text{name}(o) \mid o \in s\})} \) is a reference to the meet class of \( s \).
\[ \forall s \in \mathcal{L}, \quad \text{obj(meet(\{\text{name}(o) \mid o \in s\}) = \text{meet}(s) \}

**Rule 3:** Let \( c \) be a class and let \( a' \) be an explicit arrow class. If \( e_{\text{inh}}(c, a') \) is defined, \( r \) is a reference to \( c \), and \( n \) is the name of \( a' \) then (i) \( e_{\text{inh}}(r, n) \) is a reference to \( e_{\text{inh}}(c, a') \), and (ii) \( d(r, n) \) is a reference to \( to(e_{\text{inh}}(c, a')) \).
\[ \forall a' \in EAC, \ c \in C, \ r \in O_{ref}, \ n \in A_{nam}, \ 
\exists e_{\text{inh}}(c, a') \in AC \land \text{obj}(r) = c \land \text{name}(a') = n \implies \text{obj}(e_{\text{inh}}(r, n)) = e_{\text{inh}}(c, a') \land \text{obj}(d(r, n)) = to(e_{\text{inh}}(c, a')) \]
**Rule 4:** Let \( c \) be a class and let \( a' \) be an explicit arrow class. If \( a \text{\_inh}(c, a') \) is defined, \( r \) is a reference to \( c \), and \( n \) is the name of \( a' \) then \( a \text{\_inh}(r, n) \) is a reference to \( a \text{\_inh}(c, a') \).

\[
\forall a' \in EAC, c \in C, r \in O_{\text{ref}}, n \in A_{\text{nam}}, \\
\exists a \text{\_inh}(c, a') \in AC \wedge \text{obj}(r) = c \wedge \text{name}(a') = n \Rightarrow \text{obj}(a \text{\_inh}(r, n)) = a \text{\_inh}(c, a')
\]

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

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 6.7 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 \( S' \).

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 6.8 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:

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

27
Definition 6.9 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 6.10 Model ordering

Let \( M \) and \( M' \) be two models\(^4\) of \( P \). We say that \( M \) is less than or equal to \( M' \), denoted \( M \leq M' \), iff there is a mapping \( \mathcal{F} : O \to O' \) such that

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

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 6.11 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.

In the rest of the section, we consider only valid programs.

The following proposition relates objects to system classes.

\(^4\) Symbols of structure components with a prime, such as \( O' \), \( \text{name}' \), denote components of \( M' \).
**Proposition 6.2** Let $P$ be a program with semantics $M$. The following statements hold:

1. For each $o \in E$, it holds that either $In(o, \#\text{Indiv})$ or $In(o, \#\text{Arrow})$ or $In(o, \#\text{Hybrid})$.

2. For each $o \in E$, it holds that
   
   $o \in C$ iff either $In(o, \#\text{IndivClass})$ or $In(o, \#\text{ArrowClass})$ or $In(o, \#\text{Hybrid})$.

3. For each $o \in O$, it holds that $In(o, \#\text{Object})$.

As we have seen in section 2, each real object is either a real token or a real 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 program with semantics $M$. Let $a'$ be an arrow class and let $c$ be a subclass of $\text{from}(a')$. Then, the arrow $e_{\text{inh}}(c, a')$ is restriction subclass of $a_{\text{inh}}(c, a')$.

A consequence of the above proposition is that the arrows $e_{\text{inh}}(c, a')$ and $a_{\text{inh}}(c, a')$ 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 $e_{\text{inh}}(c, a')$ is also instance of $a_{\text{inh}}(c, a')$. Note that the two arrows have the same $\text{from}$ object. Therefore, from Risa Rule 2, every instance of $a_{\text{inh}}(c, a')$ is also instance of $e_{\text{inh}}(c, a')$.

The following proposition expresses that if an arrow $a_0$ is restriction subclass of an arrow $a_1$ then $e_{\text{inh}}(c, a_0)$ coincides with $e_{\text{inh}}(c, a_1)$, and $a_{\text{inh}}(c, a_0)$ coincides with $a_{\text{inh}}(c, a_1)$.

**Proposition 6.4** Let $P$ be a program with semantics $M$. Let $a_0, a_1$ be explicit arrow classes and let $c$ be a class. If $Risa(a_0, a_1)$ and $Isa(c, \text{from}(a_0))$ then $e_{\text{inh}}(c, a_0) = e_{\text{inh}}(c, a_1)$ and $a_{\text{inh}}(c, a_0) = a_{\text{inh}}(c, a_1)$.

For an example, refer to Figure 7(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\footnote{Also called type refinement.} 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 program with semantics $M$. Let $a'$ be an explicit class and let $c$ be a subclass of $from(a')$. If the arrow $\overline{\text{inh}}(c, a')$ is restriction subclass of an explicit arrow $a$ then $\overline{\text{inh}}(c, a') = \overline{\text{inh}}(c, a)$ and $\overline{\text{inh}}(c, a') = \overline{\text{inh}}(c, a)$.

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 program with semantics $M$. 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 7(b).
7 Comparison with Related Work

Specialization hierarchies and inheritance play an important role in knowledge representation systems based on semantic networks and frames [21, 30], as well as in object-oriented and some extensible database systems [25, 11]. 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 [3], O₂ [15], ODE [1], POSTGRES [31, 33], EXODUS [8], is given in [2].

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 \( c \) [9]. 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 [2], 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 [26, 6, 4, 28, 16], do not consider inheritance of typing properties\(^6\). Yet, in many applications, reasoning on the structural definitions of the data is a necessity [29]. A property inherited by a class provides information about the class that may not be obvious by just browsing 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 [2]).

Figure 8: Example of typing property inheritance and typechecking

We will compare our approach to typing property inheritance and typechecking with that of F-logic [24], DOT [35, 36], and QUIXOTE [39]. All three data models do not support $Isa$ relations between properties. Additionally, they support $Risa$ only implicitly, based on property labels. Specifically, let a class $c$ be subclass of a class $c'$. If two properties $a$, $a'$ of classes $c$ and $c'$ have the same label then $Risa(a, a')$ is implied. In our opinion, the $Risa$ relation between $a$ and $a'$ should be expressed explicitly by declaring $Risa(a, a')$, and not

\(^6\) These data models consider non-monotonic inheritance.
implicitly by using the same label for $a$ and $a'$. Following this reasoning, we allow $a$ and $a'$ to have the same label even if they are not related through $Risa$.

**F-LOGIC** [24]

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

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

(ii) If $c[p \Rightarrow D], c[p \Rightarrow D']$ then $c[p \Rightarrow D \cup D']$.

The fact that an object $o$ has a non-typing property $p$ and the value of $p$ is $d$ is asserted by the statement $o[p \Rightarrow d]$.

For example, in F-logic, the information in Figure 8, is expressed as follows:

* Isa relations between classes, are declared as shown in the figure.

* $Vehicle\_factory[produces \Rightarrow \{Vehicle\}]$, $Only\_car\_factory[produces \Rightarrow \{Car\}]$, $Only\_boat\_factory[produces \Rightarrow \{Boat\}]$, $x[produces\rightarrow y]$.

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 with label $produces$ then the value of the property should be an instance of both $Car$ and $Boat$. F-logic enforces the Typing constraint. Thus, in Figure 8, 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 8, assume that $Car$ is not subclass of $Vehicle$. In contrast to our data model, in F-logic, this will not cause any constraint violation and $Only\_car\_factory[produces \Rightarrow \{Car, Vehicle\}]$ will be derived.

Let $P$ be a program in our data model, without Isa declarations between arrows. If $d'$ is superclass of $to(a\_inh(c, p))$, for a class $c$ and property $p$, then, in F-logic, it holds that $c[p \Rightarrow \{d'\}]$, and vice versa.
DOT [35, 36]

The knowledge representation model DOT describes properties 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 8, the In relation should be replaced by Isa. The value of a property with label p of an object c is denoted by c.p. Property inheritance is supported by the DOT rule: If Isa(c, c') then Isa(c.p, c'.p).

In DOT, the information in Figure 8, is expressed as follows:

Isa relations between classes, are declared as shown in the figure,

Isa(Vehicle\_factory, produces, Vehicle), Isa(Vehicle, Vehicle\_factory, produces),
Isa(Only\_car\_factory, produces, Car), Isa(Car, Only\_boat\_factory, produces),
Isa(Only\_boat\_factory, produces, Boat), Isa(Boat, Only\_boat\_factory, produces),
Isa(x, Amphibious\_factory), Isa(x, produces, y), (y, x, produces).

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.

Let P be a program in our data model, without Isa declarations between arrows. Let SUP and SUB be the sets of superclasses and subclasses of to(\alphanh(c, p)), for a class c and property p. Note that SUB is the set of all subclasses of meet(SUP). We can derive that SUP coincides with the set of classes d (in our data model) such that Isa(c, p, d) holds in DOT. However, the same does not hold for SUB. If fact the set of classes d (in our data model) such that Isa(d, c, p) holds in DOT, is subset of SUB. For an example, refer again to Figure 8. In our data model, the class Amphibious is subclass of the to object of the arrow produces inherited by Amphibious\_factory (the inherited arrow is not shown in the figure). However, in DOT, Amphibious is not a specialization of Amphibious\_factory produces.

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. In contrast, in our data model and F-logic, if In(y, Car) and In(y, Boat) have not been explicitly declared then the Typing
Constraint will be violated. In addition, assume that the relation \( Isa(Car, Vehicle) \) has not been explicitly declared. Then, in DOT, the relation \( Isa(Car, Vehicle) \) will be derived. In contrast, in our data model, the Arrow Isa Constraint will be violated. In F-logic, no relation will be derived and no constraint will be violated. Obviously, in DOT, no checking of the Typing and Subtyping Constraints is needed, as they are satisfied by all programs.

**QUIXOTE [39]**

QUIXOTE is a deductive object-oriented language that supports typing property inheritance. As in DOT, the \( In \) relation is not distinguished from \( Isa \). QUIXOTE assumes that the set of explicit objects with the \( Isa \) relation forms a lattice. Property inheritance is supported by the QUIXOTE rules:

(i) if \( Isa(c, c') \) then \( Isa(c.p, c'.p) \), and
(ii) if \( Isa(c.p, d) \) and \( Isa(c.p, d') \) then \( Isa(c.p, \text{meet}\{d, d'\}) \).

Rule (i) refines the value of the inherited properties, and rule (ii) refines them further. For an example, refer Figure 8. As \( \text{meet}\{Car, Boat\} = Amphibious \), it will be derived that \( Isa(\text{Amphibious\_factory\_produces, Amphibious}) \). However, it will not be derived that \( \text{Amphibious\_factory\_produces}=\text{Amphibious} \), as in our data model. Similarly to DOT, every QUIXOTE program satisfies the Typing and Subtyping Constraints. Thus, no typechecking is needed.

As we have mentioned, F-logic, DOT, and QUIXOTE do not support \( Isa \) relations between properties. For example, in Figure 4, these data models will not derive that the value of the property \( \text{collects} \) inherited by \( \text{Rich\_painting\_collector} \) from \( \text{Painting\_collector} \) is a specialization of \( \text{Expensive\_art\_object} \). 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.

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 "≤" is a partial ordering over the models of $P$ is a partial order (up to model equivalence).

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

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 $M ≤ M'$, there is a mapping $\mathcal{F} : O → O'$ that satisfies the conditions of Definition 6.10. Additionally, as $M' ≤ M$, there is a mapping $\mathcal{F}' : O' → O$ that satisfies the conditions of Definition 6.10. First, we will show that $F'$ is the inverse of $F$. Let $o ∈ O$ and $o' = \mathcal{F}(o)$. We will show that $\mathcal{F}'(o') = o$. Assume that $\mathcal{F}'(o') = o_1$. As $M ≤ M'$, it holds that $ref(o) ⊆ ref'(o')$. As $M' ≤ M$, it holds that $ref'(o') ⊆ ref(o_1)$. Thus, $ref(o) ⊆ ref(o_1)$. As $ref(o) ≠ \emptyset$, there is $r ∈ O_{ref}$ such that $obj(r) = o$ and $obj(r) = o_1$. As $obj$ is a function, it follows that $o = o_1$.

Let $o' ∈ O'$ 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 6.10 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 $o ∈ E$ 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 6.10 that name$(o')$ is defined. Thus, $o' ∈ E'$.

Let $a ∈ EA$ and let $a' = \mathcal{F}(a)$. From the above, it follows that $a' ∈ EA'$. 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, $∀ a ∈ EA$, label$(a) = label'(\mathcal{F}(a'))$.

From condition 2 of Definition 6.10 and the fact that $\mathcal{F} = \mathcal{F}'^{-1}$ it follows that $∀ o ∈ O$, $ref(o) = ref'(\mathcal{F}(o'))$.

From condition 1 of Definition 6.10 and the fact that obj is function, it follows that $∀ s ∈ L$, $\mathcal{F}(\text{meet}(s)) = \text{meet}'(\mathcal{F}(s))$.

We will now prove the following statements:
1. ∀ c, a′ ∈ 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. ∀ c, a′ ∈ 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 6.10, it follows that ∀ c ∈ \( EM_0 \), a′ ∈ \( EA \), e\_inh(c, a′) is defined iff \( e\_inh'(\mathcal{F}(c), \mathcal{F}(a′)) \) is defined. Similarly, ∀ c ∈ \( EM_0 \), a′ ∈ \( EA \), a\_inh(c, a′) is defined iff \( a\_inh'(\mathcal{F}(c), \mathcal{F}(a′)) \) is defined.

From condition 2 of Definition 6.10, the Reference Rules, and the fact that \( obj \) is a function, it follows that (i) ∀ c ∈ \( EM_0 \), a′ ∈ \( EA \), 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 (ii) ∀ c ∈ \( EM_0 \), a′ ∈ \( EA \), 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 6.10, it follows that ∀ c, a′ ∈ O, \( \text{cand\_cl}(c, a′) \) is defined then \( \mathcal{F}(\text{cand\_cl}(c, a′)) = \text{cand\_cl}'(\mathcal{F}(c), \mathcal{F}(a′)) \). 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 ∀ a ∈ \( A \), \( \mathcal{F}(from(a)) = from'(\mathcal{F}(a)) \) and \( \mathcal{F}(to(a)) = to'(\mathcal{F}(a)) \).

From condition 4 of Definition 6.10 and the fact that \( \mathcal{F} = \mathcal{F}^{-1} \) it follows that ∀ o, o′ ∈ O, \( Rel ∈ \{ In, Isa, Risa \} \), \( Rel(o, o′) \) holds iff \( Rel'(\mathcal{F}(o), \mathcal{F}(o′)) \) holds.

It now follows that \( M \equiv M' \).

**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 $\forall c, a' \in O, \text{cand}(c, a')$ is the same in $F$ and $M$.

Assume that there exist $c$, $a'$ such that $\text{cand}(c, a')$ is different in $F$ and $M$. Thus, there is an explicit arrow $a$, such that $\text{to}(a)$ is in $\text{cand}(c, a')$ with respect to $M$ but not with respect to $F$. Thus, there a relation $\text{Isa}(e, \text{inh}(c, a'), a)$ that is not derived in $F$ and is derived in $M$ using Exact Inheritance Rule 3. This implies that there is a relation $\text{Risa}(a, \text{inh}(c', a''), a_0)$, 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 $\text{Isa}(e, \text{inh}(c, a'), a)$. However, if this is the case then $\text{Risa}(e, \text{inh}(c', a''), a_0)$ should have been derived in $F$ using Exact Inheritance Rule 3 and Risa Rule 6. From $\text{Risa}(e, \text{inh}(c', a''), a_0)$, the relation $\text{Isa}(e, \text{inh}(c, a'), a)$ would have been derived in $F$, in the same way $\text{Isa}(e, \text{inh}(c', a'), a)$ is derived from $\text{Risa}(a, \text{inh}(c', a''), a_0)$ in $M$. However, this is impossible because we have assumed that $\text{Isa}(e, \text{inh}(c, a'), a)$ is not derived in $F$.

Thus, $M$ satisfies all inference rules. Due to conditions 1 and 2 of Definition 6.5 (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 of $P$. We will show that $M \leq M'$.

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 $F : O \rightarrow O'$ that maps every object $o \in O$ to an object $o' \in O'$ such that $\text{ref}(o) \cap \text{ref}'(o') \neq \emptyset$. From the fact that $\text{obj}$ is a function, there is only one such $o'$. Obviously, $F$ satisfies conditions 1 and 2 of Definition 6.10.

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

We will now show the following statements:

---

Note that Exact Inheritance Rule 3 is a special case of Risa Rule 5 (arrow $a_1$ in Risa Rule 5 is replaced by $e, \text{inh}(c, a')$ and the condition on the to objects is eliminated).

Symbols of structure components with a prime, denote components of $M'$.  

39
1. \( \forall c, a' \in O, \) if \( e \cdot \text{inh}(c, a') \) is defined then \( \mathcal{F}(e \cdot \text{inh}(c, a')) = e \cdot \text{inh} \left( \mathcal{F}(c), \mathcal{F}(a') \right) \) and 
   \( \mathcal{F} \left( \text{to}(e \cdot \text{inh}(c, a')) \right) = \text{to}'(e \cdot \text{inh} \left( \mathcal{F}(c), \mathcal{F}(a') \right)) \)

2. \( \forall c, a' \in O, \) if \( a \cdot \text{inh}(c, a') \) is defined then \( \mathcal{F}(a \cdot \text{inh}(c, a')) = a \cdot \text{inh} \left( \mathcal{F}(c), \mathcal{F}(a') \right). \)

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

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

Obviously, the previous statements now hold if we replace \( EM_0 \) by \( EM_1 = e \cdot \text{inh}(EM_0, EA) \cup a \cdot \text{inh}(EM_0, EA) \). 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 6.10. Thus, Theorem 6.1 follows. \( \Box \)

**Proposition** Let \( P \) be a program with semantics \( M. \) All derived objects and relations according to \( M \) represent corresponding real objects and relations holding in the real world.

**Proof** Soundness of the Isa, Risa, and Exact Inheritance Rules is proved in . Soundness of the remaining inference rules can be proved in the same manner.

**Proposition 6.2** Let \( P \) be a program with semantics \( M. \) The following statements hold:

1. For each \( o \in E, \) it holds that either \( \text{In}(o, \#\text{Indiv}) \) or \( \text{In}(o, \#\text{Arrow}) \) or \( \text{In}(o, \#\text{Hybrid}). \)

2. For each \( o \in E, \) it holds that 
   \( o \in C \) iff either \( \text{In}(o, \#\text{IndivClass}) \) or \( \text{In}(o, \#\text{ArrowClass}) \) or \( \text{In}(o, \#\text{Hybrid}). \)

3. For each \( o \in O, \) it holds that \( \text{In}(o, \#\text{Object}). \)
Proof

**Statement 1:** As $M$ is the least model of $P$, each object in $E$ has been created through a declaration $\text{indiv}()$, $\text{hybrid}()$, or $\text{arrow}()$. Therefore, $E \subseteq I \cup H \cup A$. Statement 1 now follows directly from Domain Rule 3.

**Statement 2:** [Forward direction] Let $o \in C$. From Domain Rule 3, it follows that $\text{In}(c, \#\text{Class})$. From statement 1 and Domain Rules 4 and 5, it follows that $o$ in an instance of $\#\text{IndivClass}$ or $\#\text{ArrowClass}$, or $\#\text{Hybrid}$.

[Backward direction] Let $o$ in an instance of $\#\text{IndivClass}$ or $\#\text{ArrowClass}$, or $\#\text{Hybrid}$. Then, from the system declarations, it follows that $o$ is an instance of $\#\text{Class}$. From Domain Rule 3, it follows that $o \in C$.

**Statement 3:** First assume that $o \in E$. Then $\text{In}(o, \#\text{Object})$ follows from Statement 1 and the facts that $\text{Isa}(\#\text{Indiv}, \#\text{Object})$, $\text{Isa}(\#\text{Arrow}, \#\text{Object})$, and $\text{Isa}(\#\text{Hybrid}, \#\text{Object})$ (following from the system declarations). If $o \notin E$ then $o$ is a derived objects. Thus, $o \in C$. From Domain Rule 3, it follows that $\text{In}(o, \#\text{Class})$. As $\text{Isa}(\#\text{Class}, \#\text{Object})$, it follows that $\text{In}(o, \#\text{Object})$.  

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

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

**Proposition 6.4** Let $P$ be a program with semantics $M$. Let $a_0, a_1$ be arrow classes and $c$ be a class. If $\text{Risa}(a_0, a_1)$ and $\text{Isa}(c, \text{from}(a_0))$ then $e\text{\_inh}(c, a_0) = e\text{\_inh}(c, a_1)$ and $a\text{\_inh}(c, a_0) = a\text{\_inh}(c, a_1)$.

**Proof** We will first prove that $e\text{\_inh}(c, a_0) = e\text{\_inh}(c, a_1)$. As $\text{Risa}(e\text{\_inh}(c, a_1), a_1)$ (from Exact Inheritance Rule 2) and $\text{Risa}(a_0, a_1)$ (from hypothesis), it follows from Exact Inher-
itance Rule 3 that $\text{Isa}(e, \text{inh}(c, a_1), a_0)$. As $\text{Risa}(e, \text{inh}(c, a_0), a_0)$, it follows from Exact Inheritance Rule 3 that $\text{Isa}(e, \text{inh}(c, a_1), e, \text{inh}(c, a_0))$.

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

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

\[ \diamond \]

**Proposition 6.5** Let $P$ be a program with semantics $M$. Let $a'$ be an explicit class and let $c$ be a subclass of from($a'$). If the arrow $a, \text{inh}(c, a')$ is restriction subclass of an explicit arrow $a$ then $e, \text{inh}(c, a') = e, \text{inh}(c, a)$ and $a, \text{inh}(c, a') = a, \text{inh}(c, a)$.

**Proof** We will first prove that $e, \text{inh}(c, a') = e, \text{inh}(c, a)$.

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

We will now prove that $a, \text{inh}(c, a') = a, \text{inh}(c, a)$. As $e, \text{inh}(c, a') = e, \text{inh}(c, a)$, it follows that $\text{cand.cl}(c, a') = \text{cand.cl}(c, a)$. Therefore, $\text{to}(a, \text{inh}(c, a')) = \text{to}(a, \text{inh}(c, a))$. From hypothesis, it follows that $\text{Risa}(a, \text{inh}(c, a'), a)$. From Approximate Inheritance Rule 2, it holds that $\text{Risa}(a, \text{inh}(c, a), a)$. Thus, from Risa Rule 5, it follows that $\text{Isa}(a, \text{inh}(c, a'), a, \text{inh}(c, a))$. Again, from Risa Rule 5, it follows that $\text{Isa}(a, \text{inh}(c, a'), a, \text{inh}(c, a))$. Again, from Risa Rule 5, it follows that $\text{Isa}(a, \text{inh}(c, a'), a, \text{inh}(c, a))$.

42
\( Isa(a, \text{inh}(c, a)), \ a, \text{inh}(c, a') \) \). Therefore, it follows from Approximate Inheritance Rule 3 that \( a, \text{inh}(c, a') = a, \text{inh}(c, a) \). \( \diamond \)

**Proposition 6.6** Let \( P \) be a program with semantics \( M \). 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 \( Isa(d, \text{to}(a)) \), the Exact Inheritance Rule 4 should have been used. This implies that the \( d \) should be the to object of an exact inherited arrow \( e, \text{inh}(c, a') \). However, this is impossible because \( d \) is an explicit class or meet class. \( \diamond \)
References


45