On formalization of object model by unifying intensional and extensional representations

Iztok SAVNIK
Faculty of Mathematics, Natural Sciences and Information Technologies, University of Primorska, Glagoljaška 8, 5000 Koper, Slovenia

Abstract. In this paper we present the consequences of unifying the representation of the schema and the instance levels of an object programming language to the formal representation of object model. The uniform representation of schema and instance levels of object languages is achieved, as in the frame-based knowledge representation languages [9], by representing them using a uniform set of modeling constructs. We show that, using such an approach, the structural part of the object language model can be described in a clear manner providing the simple means for the description of the main constructs of the structural model and the relationships among them. Further, we study the consequences of releasing the boundary between the schema and the instance levels of an object programming language by allowing the definition of objects which include data from both levels. We show that few changes are needed in order to augment the previously presented formal definition of the structural part of object language to represent the extended object model.

Keywords. programming languages, object model, conceptual models, language semantics.

1. Introduction

In spite of considerable research effort directed to the problems of the formalization of object model in the areas of programming languages [8,14,13], databases [15,2,12,1,4] and conceptual modeling [3,5,6] in the last few decades, there is still a lack of a precise theoretical framework for object model. The main reason for this lies in the rich set of sophisticated data modeling constructs provided by the numerous variants of object models.

In this paper we focus on one aspect of object model: *a structural model.* Formal semantics of object model is presented in the denotational style [23]. The presented formalization unifies the schema and the instance levels of a language by treating classes as objects in a similar way as frames [9] are used to represent abstract concepts.

The idea of treating classes as objects is present in various object-oriented programming languages. For instance, Smalltalk [10] treats every construct of the language as object. Programming language C++ [20] includes some introspective language constructs that provide the access to some properties of classes. Finally, in Java [11] classes are indeed treated as objects but have different access mechanisms comparing to the access to ground objects.

We show that the uniform treatment of extensional and intensional parts of object model allows clear and robust definition of the formal representation of object model. The same modeling constructs are used for the representation of extensional and intensional parts of an object model. In the beginning we retain a strict separation between the conceptual schema and the instance parts of object repository. In the sequel, we observe the consequences of releasing the boundary between these two parts of a object repository. Merging the intensional and extensional levels of an object programming environment is achieved by allowing objects to include individual objects *and* classes as their components.

Uniform treatment of structural model simplifies the representation of complex environments that have rich conceptual schemata. Examples include Internet databases and repositories, and distributed database environments. Such environments usually contain rich metadata repository where the distinction between extensional and intensional representation is often blurred. Apart from serving as formal framework for structural model, uniform treatment of extensional and intensional parts of object model can serve as a basis for the definition of uniform operations for the manipulation of intensional and extensional parts of object repositories. They are presented in [16].

The first part of formalization, presenting the formal treatment of identifiers and values, is given in Sections 2 and 3. We start the formal presentation by describing identifiers and their structural properties. We define values, relate them to the previously presented identifiers and describe their properties. Section 4 presents formal definition of objects and their properties. The properties of objects are derived from the properties of values and identifiers. In Section 5 we remove the boundary between the schema and the instance levels of object model and study the consequences of this in the framework of the previously presented formalisation. Related work is presented in Section 6. Finally, concluding remarks and some aspects of implementation of the presented model are given in Section 7.

2. Identifiers

Let us first define some basic terminology used in the paper. We assume the existence of a predefined infinite set of identifiers $\mathcal{O}$. An *identifier* is a unique symbol which represents an abstract or concrete entity from the real world. Identifiers will be denoted by terms written in the lower-case letters. For example, the identifier *tom* serves as a unique identification of a person whose name is "Tom", or, the identifier *student* stands for the unique identification of the abstract representation of a student.

The set $\mathcal{O}$ is further divided into the set of individual identifiers $\mathcal{O}_D$ representing concrete entities such as persons, and the set of class identifiers $\mathcal{O}_C$ representing abstract concepts, which usually stand for a group of individual entities. In some cases we will refer to the individual and class identifiers simply as individuals and classes.

The most significant difference between class identifiers and individual identifiers is in their *interpretations*. While the interpretation of an individual identifier is the individual itself, the interpretation of a class identifier is the set of individuals.

Definition 1. *Let $c \in \mathcal{O}_C$. The interpretation of c, denoted $\Pi[\![c]\!]$, has the following properties: (i) $\Pi[\![c]\!] \subset \mathcal{O}_D$, and (ii) $\forall p(p \in \mathcal{O}_C \wedge p \neq c \Rightarrow \Pi[\![c) \cap \Pi[\![p]\!] = \emptyset)$.*

The class interpretation specifies the *membership* relationship among individual and class identifiers. Let $id_1 \in \mathcal{O}_D$ and $id_2 \in \mathcal{O}_C$. The identifier id_1 is a member of the identifier id_2 if $id_1 \in \Pi[\![id_2]\!]$. The membership relationship should not be exchanged with the *instantiation* relationship, which is defined shortly.

A binary relation among class identifiers, denoted as $(id_1 \ subclass \ id_2)$ where $id_1, id_2 \in \mathcal{O}_C$, is used to represent the inheritance hierarchy of classes. We assume that this relation is given by the definition of the conceptual schema of an object-oriented database. Using the *subclass* relationship, we define a relationship $\preceq_i$.

Definition 2. *Let $id_1, id_2 \in \mathcal{O}$ then $id_1 \preceq_i id_2$ if one of the following holds: (i) $id_1 = id_2$, (ii) $id_1, id_2 \in \mathcal{O}_C \Rightarrow \exists id_3(id_3 \in \mathcal{O}_C \wedge (id_1 \ subclass \ id_3) \wedge id_3 \preceq_i id_2)$, or (iii) $id_1 \in \mathcal{O}_D \wedge id_2 \in \mathcal{O}_C \Rightarrow \exists id_3(id_3 \in \mathcal{O}_C \wedge id_1 \in \Pi[\![id_3]\!] \wedge id_3 \preceq_i id_2)$.*

The relationship $\preceq_i$ is called *more_specific* or, the opposite, *more_general* relationship. It can be easily seen that the relationship $\preceq_i$ organises identifiers into the partially ordered set (abbr. *poset*). It is reflective, that is, $id \preceq_i id$ for all $id \in \mathcal{O}$. It is antisymmetric since $id_1, id_2 \in \mathcal{O} \wedge id_1 \preceq_i id_2 \wedge id_2 \preceq_i id_1$ implies $id_1 = id_2$. It is also transitive since $id_1 \preceq_i id_2 \wedge id_2 \preceq_i id_3$ implies $id_1 \preceq_i id_3$ for $id_1, id_2, id_3 \in \mathcal{O}$.

Lemma 1. *The set $\mathcal{O}$ is partially ordered by the relationship $\preceq_i$.*

The ordinary class interpretation maps a class identifier to a set of individual identifiers called the members of the class. By taking into account the previously defined partial ordering among the class and individual identifiers, another interpretation is introduced. The *inherited interpretation* [2] of the class identifier c includes the members of the class c and the members of class c's subclasses.

Definition 3. *Let $c \in \mathcal{O}_C$. The inherited interpretation of c, denoted $\Pi^*[\![c]\!]$, is defined as: $\Pi^*[\![c]\!] = \bigcup_{p \in \mathcal{O}_C \wedge p \preceq_i c} \Pi[\![p]\!]$*

Using the above definition of the inherited interpretation, we define the *instantiation* relationship commonly used to represent the associations between individual and class concepts. Let $id_1 \in \mathcal{O}_D$ and $id_2 \in \mathcal{O}_C$. The identifier id_1 is an instance of id_2 if $id_1 \in \Pi^*[\![id_2]\!]$.

3. Values

The concept of identifier is extended to the notion of *value*. The set of all values is denoted as $\mathcal{V}$. We distinguish between two basic types of value: *identifiers* and *structured values*. The *set* and *tuple* constructors are used to build the structured values from identifiers. Structured values are divided into the set of ground values $\mathcal{V}_D$, and the set of values $\mathcal{V}_T$ that represent types. Further, we assume the existence of a set of attribute names $\mathcal{A}$. The following definition states the syntactical structure of the values.

Definition 4. *The value is one of the following: (i) id* $\in \mathcal{O}$*, (ii)* $\{o_1, \ldots, o_n\}$*, where* $o_i \in \mathcal{V}$*, or (iii)* $\langle A_1 : o_i, \ldots, A_n : o_n \rangle$*, where* $o_i \in \mathcal{V}$ *and* $A_i \in \mathcal{A}$*.*

Values which include only the individual identifiers are called *ground values*. When the values are composed solely of class identifiers, we refer to them as *types*. Analogously to our perception of class identifiers, types stand for the abstract representation of a set of values. Formal definition of a type is given in [19].

Close integration of the concepts of class identifier and type provides a clear method for the definition of type interpretation which can be now defined as a straightforward extension of the class interpretation. The type interpretation is defined as follows. Note that in the following definition Π_c^* denotes the inherited interpretation of class identifiers.

Definition 5. *Let* $t \in \mathcal{V}_T$*. With respect to type* t *structure, its interpretation, denoted* $\Pi[\![t]\!]$*, is: (i)* $t \in \mathcal{O}_C \Rightarrow \Pi[\![t]\!] = \Pi_c^*[\![t]\!]$*, (ii)* $t = \{s\} \Rightarrow \Pi[\![t]\!] = \{o; o \subset \Pi[\![s]\!]\}$*, or, (iii)* $t = \langle A_1 : t_1, \ldots, A_n : t_n \rangle \Rightarrow \Pi[\![t]\!] = \{\langle A_1 : v_1, \ldots, A_n : v_n \rangle; v_i \in \Pi[\![t_i]\!]\}$*.*

This definition of the type interpretation specifies the *membership* relationship between ground values and types.

The values that are more specific, or "below" in the ordering defined by the relationship $\preceq_v$, refine the more general values that are "higher" in the set of values $\mathcal{V}$ with regard to the relationship $\preceq_v$. The following definition states the syntactical definition of $\preceq_v$.

Definition 6. *Let* $v_1, v_2 \in \mathcal{V}$ *be values. The value* v_1 *is more specific then the value* v_2*, denoted by* $v_1 \preceq_v v_2$*, if one of the following holds: (i)* $v_1, v_2 \in \mathcal{O} \Rightarrow v_1 \preceq_i v_2$*, (ii)* $v_1, v_2 \in \mathcal{V}_T \wedge v_1 = \{s\} \wedge v_2 = \{t\} \Rightarrow s \preceq_v t$*, (iii)* $v_1, v_2 \in \mathcal{V}_T \wedge v_1 = \langle A_1 : a_1, \ldots, A_n : a_n \rangle \wedge v_2 = \langle B_1 : b_1, \ldots, B_k : b_k \rangle \Rightarrow n \geq k \wedge \forall b_i (b_i \Subset v_2 \Rightarrow (A_i = B_i \wedge a_i \preceq_v b_i))$*, or (iv)* $v_1 \in \mathcal{V}_D \wedge v_2 \in \mathcal{V}_T \Rightarrow v_1 \in \Pi[\![v_2]\!]$*.*

The relationship $\preceq_v$ forms a partial ordering of values. It can be easily seen that it is reflective, antisymmetric and transitive.

Lemma 2. *The set* $\mathcal{V}$ *is partially ordered by the relationship* $\preceq_v$*.*

The previous definition of the value poset captures the notion of partial ordering of types as defined by Cardelli in [8], or Vandenberg in [22]. It can be obtained by restricting the set of all values $\mathcal{V}$ to types $\mathcal{V}_T$.

The type interpretation defined in the previous sub-section maps a type T to a set of its members whose structure is strictly the same as the structure of a given type. We remove this constraint by defining the *inherited interpretation* of a type to be the union of the interpretation of a given type and the interpretations of all types which are more specific than a given type.

Definition 7. *Let $t \in \mathcal{V}_T$. The inherited interpretation of t, denoted $\Pi^*[\![t]\!]$, is defined as:* $\Pi^*[\![t]\!] = \bigcup_{s \in \mathcal{V}_T \wedge s \preceq_v t} \Pi[\![s]\!]$.

The above definition captures the notion of the *instantiation* relationship between ground values and types. Formally, the instantiation relationship can be defined as follows. Let $v \in \mathcal{V}_D$ and $t \in \mathcal{V}_T$. The value v is an instance of type t if $v \in \Pi^*[\![t]\!]$.

Definition 6 gives a syntactical means for checking the relationship $\preceq_v$ between values. The following theorem shows the correspondence between the syntactical definition of $\preceq_v$ and the inherited interpretation function Π^*.

Theorem 1. *Let $t_1, t_2 \in \mathcal{V}_T$. The following relation between the relationship $\preceq_v$ and the interpretation Π^* holds: $t_1 \preceq_v t_2 \iff \Pi^*[\![t_1]\!] \subseteq \Pi^*[\![t_2]\!]$.*

Proof. The first part of the proof is to show that the syntactical definition implies the subsumption of the corresponding inherited interpretations. Definition 7, which presents the inherited interpretation of types, uses the relationship $\preceq_v$ to identify more specific values. If t_1 is more specific than t_2 then, according to Definition 7, the set $\Pi^*[\![t_1]\!]$ has to be included in the set $\Pi^*[\![t_2]\!]$.

The reverse direction can be proved in a similar manner. Suppose the relationship $\Pi^*[\![t_1]\!] \subseteq \Pi^*[\![t_2]\!]$ holds. Definition 7 states that $\Pi^*[\![t_2]\!]$ includes all inherited interpretations of more specific types from $\mathcal{V}_T$. Therefore, if $\Pi^*[\![t_1]\!]$ is included in $\Pi^*[\![t_2]\!]$, than t_1 has to be in the set of types which are more specific than t_2, or $t_1 \preceq_v t2$. □

4. Object Model

The proposed data model distinguishes between two aspects of objects. First, every object has an identity, also called object identity (oid) which is realised by an identifier that distinguishes it from all other objects in the language repository. Second, every object has a value which describes its state. The two basic object aspects are connected by means of a *value assignment function* that maps identifiers to corresponding values.

We distinguish between *primitive* and *defined* objects. The identity of the primitive object is the same as its value. The value of the defined object can be any of the previously defined values.

Definition 8. *An object is a pair $o = (id, v)$, where $id \in \mathcal{O}$ and $v \in \mathcal{V}$. The object can be in the following two forms: (i) primitive object (id, id), where $id \in \mathcal{O}_D$, or (ii) defined object: (id, v), where $id \in (\mathcal{O} - \mathcal{O}_D) \wedge v \in \mathcal{V}$.*

The *individual object* represents a single concrete entity from a modelling environment. Its value includes solely the individual identifiers. Secondly, *class object* represents an abstract entity which stands for: the representation of an abstract concept, or, from the other point of view, an abstract representation of the set of individual objects. The value of a class object includes solely the class identifiers. Examples of individual and class objects are given in [19].

4.1. Relations between ids, values and objects

In this sub-section, we present in more detail some relationships among the basic elements of formal presentation: identifiers, values and objects. First, we present the value assignment functions. Next, the inheritance of properties in the context of presented formal view is discussed. Finally, we present the relations between partially ordered sets defined in previous sections and the partially ordered set of objects.

An object is described by a set of properties which are represented by attributes. We assume that the attributes are defined for a particular object at the time of their creation. The individual objects inherit attributes from their parent class objects. The attributes of class objects are defined by the definition of classes in a programming language environment.

The attributes that correspond to an object consist of: the attributes that are directly associated to the class, and the inherited attributes. For this purpose, two assignment functions are defined. The *value assignment* function ν is formally defined as follows.

Definition 9. *Let* $id \in \mathcal{O}$ *and* $\nu{:}\mathcal{O} \rightarrow \mathcal{V}$ *a value assignment function such that* $\nu(id) = v$ *where* $v \in \mathcal{V}$ *and* (id, v) *is an object.*

The *inherited value assignment* function ν^* returns all properties of an object o. It is defined by the following definition. The relationship $\Subset$ denotes the component-of relationship.

Definition 10. *Let* $id \in \mathcal{O}$*. The inherited value assignment function* $\nu^* : \mathcal{O} \rightarrow \mathcal{V}$ *is defined as:* $\nu^*(id) = \langle A_1{:}v_1, \ldots, A_k{:}v_k \rangle$*, where* $\forall A_i(A_i \in Atr \wedge \exists p(id \preceq_i p \wedge A_i{:}v_i \Subset \nu(p)))$*.*

4.2. Structural inheritance

The inherited value assignment function ν^* (Definition 10) can return a value which includes more than one attribute with the same name. A problem arises when we would like to access the value of such an attribute. There are two types of conflicts. In the first case, inherited attributes with the same name are defined for objects related by the relationship $\preceq_i$. In the second case, the cause of conflict is multiple inheritance. In this situation, an object inherits two or more attributes with the same name from more general objects which are not related by the relationship $\preceq_i$.

The first type of conflict is resolved using the *overriding* principle; the attribute which is the closest with respect to the poset of identifiers is chosen. Still, according to Definition 10, both attributes are defined for the particular object. The value of overridden attribute can be accessed by explicitly stating the object of its definition. This approach is used in C++ as well as in Java.

An additional property of overriding principle is required in proposed data model to establish conditions for partial ordering of objects. The value of attribute A defined for a class c must be more specific than the value of attribute A defined by a superclass of c. The attribute of subclass *overrides* the attribute of superclass.

Invariant 1. *(Component refinement) Let $id_1, id_2 \in \mathcal{O}$, $A{:}v_1 \Subset \nu(id_1)$ and $A{:}v_2 \Subset \nu(id_2)$. The following implication must hold: $id_1 \preceq_i id_2 \Rightarrow \nu^*(id_1).A \preceq_o \nu^*(id_2).A$ (or $v_1 \preceq_i v_2$).*

The symbol $\Subset$ stands for the relationship component-of. The value assignment functions ν and ν^* are used to obtain the values of identifiers. The dot operator is then used to select the value of attribute A.

The property expressed by the above definition is necessary for the definition of partial ordering relationship $\preceq_o$ and for the definition of static type checking algorithm [16].

Let us now show the second type of conflict that can appear with inheritance. If a class inherits from two super-classes which are not related by the inheritance hierarchy, two attributes and/or methods with the same name can appear in the description of this class. This feature is usually called *multiple inheritance* [21,8,14].

In the presented work we do not deal with this problem. The user has to state the class where the attribute or the method is defined by means of explicit quantification. Similar approach has been taken in the implementation of C++ [21]. Quite differently, in Java implementations multiple inheritance of classes is forbiden [11].

4.3. The structures of objects

The partially ordered set of values can be seen as the following two posets. First, the relationship $\preceq_i$ organizes the identifiers into a partially ordered set. Second, values that correspond to each of the identifiers are partially ordered by the relationship $\preceq_v$. In other words, if the relationship $\preceq_i$ holds between identifiers, then the relationship $\preceq_v$ holds among the corresponding values. This is expressed by the following Lemma. Note that id_1 and id_2 can be individual or class identifiers.

Lemma 3. *Let $id_1, id_2 \in \mathcal{O}$ such that $id_1 \preceq_i id_2$, then $\nu^*(id_1) \preceq_v \nu^*(id_2)$.*

Proof. When the inherited value assignment ν^* is applied to an identifier id, it returns the union of the attributes that are defined for the object referenced by id and all its more general objects. Since $id_1 \preceq_i id_2$ and since Invariant 1 requires that the attribute is always overridden by a more specific attribute, we can conclude that $\nu^*(id_1) \preceq_v \nu^*(id_2)$. □

As a consequence of the above Lemma, we can define a relationship among objects which integrates the relationships $\preceq_i$ and $\preceq_v$. Analogously to the relationships $\preceq_i$ and $\preceq_v$, we denote the relationship $\preceq_o$ and we call it the relationship *more_specific* defined on objects.

Lemma 4. *(relationship $\preceq_o$) Let $o_1 = (id_1, v_1)$ and $o_2 = (id_2, v_2)$ be objects. The object o_1 is more specific than o_2, or $o_1 \preceq_o o2$, iff $id_1 \preceq_i id_2$.*

Proof. By Lemma 3. □

Since the object identifiers uniquely identify objects and since, by the above Lemma, the partial ordering relationship $\preceq_v$ among object values is determined by the relationship $\preceq_i$ among object identifiers, the complete set of objects is also partially ordered by the relationship $\preceq_o$.

Lemma 5. *The set of objects* $\{(id, v); id \in \mathcal{O} \wedge v = \nu(id)\}$ *is partially ordered by the relationship* $\preceq_o$.

In a similar way to the above definition of the relationship $\preceq_o$ and partial ordering of objects, the *ordinary interpretation* and the *inherited interpretation* of class identifiers can be extended to class objects. They are denoted by Π and Π^*.

Definition 11. *Let* $o_c = (id_c, v_c)$, *where* $id_c \in \mathcal{O}_C$ *and* $v_c \in \mathcal{V}_T$, *be a class object. The interpretation of* o_c, *denoted* $\Pi(o_c)$, *is:* $\Pi(o_c) = \{(id, v); id \in \Pi[\![id_c]\!] \wedge v = \nu^*(id)\}$.

The inherited interpretation of class objects can then be defined in the same manner. It is based on poset $(\mathcal{O}, \preceq_i)$.

Definition 12. *Let* $o_c = (id_c, v_c)$, *where* $id_c \in \mathcal{O}_C$ *and* $v_c \in \mathcal{V}_T$, *be a class object. The interpretation of* o_c, *denoted* $\Pi^*(o_c)$, *is:* $\Pi^*(o_c) = \{(id, v); id \in \Pi^*[\![id_c]\!] \wedge v = \nu^*(id)\}$.

Again, as a consequence of the unique identification of objects by means of object identifiers, the *membership* and the *instantiation* relationships between the class objects and the individual objects are defined in the same manner as the membership and instantiation relationships among the class and individual identifiers.

5. Releasing the boundary

In the previous sections we have retained strict boundary between schema and instance levels of object repository; this was achieved by making a strict distinction between values which represent *types* and *ground values.* The components of the former are solely the class individuals, while the components of the later are solely the individual identifiers. In this section we study consequences of allowing values to include individual *and* class identifiers. We refer to such values as *abstract values.*

The structure and the properties of identifiers is not affected by the change in the definition of values. Therefore, we study the consequences of mixing schema and instance levels of object repository by revising the properties of values, and, further, the properties of objects.

5.1. Values

We start with the definition of values given by Definition 4. The definition does not restrict the structure and the contents of values in any way. The set and tuple structured values can include individual and class identifiers as leaf components.

Example 1. *As an example, the value $\langle name{:}string, age{:}\ int, works\_at{:}cs\rangle$ is an abstract value describing the structure of values representing the employees of Computer Science Department represented by an identifier cs. Note that string and int are class identifiers while cs is an individual identifier.*

The partial ordering relationship $\preceq_v$ defined in Section 3 has to be redefined to relate values which are composed of individual and class identifiers. Intuitively, the structured value v_1 is more specific than value v_2 if every component of v_2 is replaced by a more specific or equal component. Formally, the relationship *more_specific* $\preceq_v$ on abstract values is defined as follows.

Definition 13. *Let $v_1, v_2 \in \mathcal{V}$. The value v_1 is more specific then the value v_2, denoted as $v_1 \preceq_v v_2$, if one of the following holds: (i) $v_1 \in \mathcal{O} \wedge v_2 \in \mathcal{O}$: $v_1 \preceq_i v_2$, (ii) $v_1 = \{s_1, \ldots, s_n\} \wedge v_2 = \{t_1, \ldots, t_k\}$: $\forall t_i(t_i \in v_2 \wedge \exists s_j(s_j \in v_1 \Rightarrow s_j \preceq_i t_i))$, or (iii) $v_1 = \langle A_1{:}a_1, \ldots, A_n{:}a_n\rangle \wedge v_2 = \langle B_1{:}b_1, \ldots, B_k{:}b_k\rangle$: $\forall b_i(B_i{:}b_i \Subset v_2 \wedge !\forall a_j(A_j{:}a_j \Subset v_2 \wedge A_j = B_i \Rightarrow a_j \preceq_o b_i))$.*
The symbol $\Subset$ stands for the relationship component-of and the augumented quantifier $!\forall$ means $\forall$ but at least one.

Definition 6 poses more constraints on the structure defined by relationship $\preceq_v$ than Definition 13 since it is tied to the object model of programming systems. The only time it refers to the ground values is when describing relationship between the instances and their types: $v_1 \in \mathcal{V}_D \wedge v_2 \in \mathcal{V}_T \Rightarrow v_1 \in \Pi[\![v_2]\!]$. The relationship $\preceq_v$ is more complex in the case of abstract values ordered by Definiton 13. Let us present some examples of pairs of values for which the relationship $\preceq_v$ holds.

Example 2. *Suppose object repository includes: a set of class identifiers student, phd_student, etc.; a set of individual identifiers s_1, s_2, s_3, etc. which are the members of class student; a set of identifiers e_1, e_2, etc. representing employees; etc. The following relationships are valid: $s_1 \preceq_v student$, $\{s_1, s_2, s_3\} \preceq_v \{student\}$, $\{phd\_student, e_1, e_2\} \preceq_v \{student, employee\}$, and $\langle name{:}str, age{:}int, lives{:}\ addr\rangle \preceq_v \langle name{:}str, age{:}int\rangle$.*

Similarly to the values presented in Section 3, the set of values which can include individual and class identifiers is partially ordered.

Lemma 6. *The set of values $\mathcal{V}$ is partially ordered by the relationship $\preceq_v$.*

Proof. We have to show that reflexivity, antisymmetry and transitivity properties hold for the relationship $\preceq_v$. We consider here only transitivity. Firstly, the identifies are partially ordered by Lemma 1. Let v_1, v_2 and v_3 be sets such that $v_1 \preceq_v v_2$ and $v_2 \preceq_v v_3$. If for each element of v_3 there exists a set of more specific elements from v_2 for which, in turn, there exists a set of elements of v_1, then it is also true that for each element of v_3 there exists a set of more specific elements of v_1. Hence, $v_1 \preceq_v v_3$ holds. The case when v_1, v_2 an v_3 are tuples can be proved in a similar manner. □

The interpretation of values Π given by Definition 5 and the interpretation Π^* presented by Definition 7 do not need to be changed to express the interpretations for the newly defined values. However, these two interpretations do not express the complete semantics of mixed values. Using the previously defined partial ordering relationship $\preceq_v$, we define another type of interpretation. For a given value v the newly defined interpretation, referred to as *natural interpretation*, includes all more specific values of v. Such interpretation allows the variables to range over individual values and types.

Definition 14. *Let $v \in \mathcal{V}$. Natural interpretation $\Pi^\diamond(v)$ is defined as follows: $\Pi^\diamond[\![v]\!] = \{v'; v' \in \mathcal{V} \wedge v' \preceq_v v\}$.*

5.2. Objects

In this sub-section, we revise the relationships among objects presented in Section 4, and point out some differences which are the consequences of different definitions of values.

The definition of objects given by Definition 8 can withstand the change in the definition of values. To reconcile, each object has a unique identifier and a value which can now include individual and class identifiers. We differentiate between *ground objects* whose values are composed solely of individual identifiers, and *abstract objects* whose values include at least one class identifier.

The inheritance principle defined for ordinary objects is used for objects with mixed data and schema as well. As for ordinary objects presented in Section 4, the value of object can be obtained as presented by the definition of inherited value assignment function ν^* (Definition 10). The properties which values are ground are in the context of abstract objects inherited to all subclasses and instances, but can not be refined within more specific objects. Different semantics is used in Java where such attributes are called *static data members (properties)* [11]. In this case, the properties that have ground values are not inherited. Similar design was chosen in Semantic Data Model [12] where such properties are called *class attributes* and they are used to describe a property of classes, solely. To subsume both semantics, Kifer in [14] proposes the use of two kinds of properties: inheritable and non-inheritable.

The interpretations Π and Π^* of ordinary class objects presented in Definitions 11 and 15 can be adopted for mixed objects without any changes. These interpretations remain to include only ground objects. Finally, the *natural interpretation* of types (Definition 16) can be used also for mixed objects.

Definition 15. *Let $o = (id, v)$, where $id \in \mathcal{O}$ and $v \in \mathcal{V}$, be a class object. The natural interpretation of o, denoted $\Pi^\diamond(o)$, is: $\Pi^*(o) = \{(id', v'); id' \preceq_i id \wedge v' = \nu^*(id')\}$.*

The natural interpretation of an object includes besides the ground objects also *abstract objects* including pure "type objects" as well.

Definition 16. *Let $o \in \mathcal{O}$ and $(o, \nu^*(o))$ is an object. The natural interpretation $\Pi^\diamond(w)$ is defined as follows. $\Pi^\diamond[\![t]\!] = \{v; v \preceq_o w\}$.*

6. Related work

In this section, we overview the existing formalisations of object models and the representation languages that are related to, or have influenced on the design of the presented formalization of object model. A more complete overview of related work can be found in [19].

First of all, the presented formal treatment of objects bears close resemblance to the Frame-based languages [9]. The formal view of object model is based on ideas introduced by Frame Logic (abbr. F-Logic) [14]. In comparison to F-Logic, the presented formalization proposes a view of object which is closer to recent implementations of object programming systems. We define the semantics of object model that is close to the view presented in [2,15], show the consequences of treating classes in the same manner as individual objects, and, afterwards, release the barrier between schema and individual objects by allowing values to include individual and class identifiers.

The proposed formalisation uses many ideas presented by some existing formal representations of the object-oriented database model. Firstly, the formalisation is closely related to the formalisation of the O_2 database model proposed by Lecluse at al. in [15], and to the formal presentation of the database model of IQL [2]. Secondly, our work is related to the formalisation of the database model EXTRA presented by Vandenberg in [22]. Among the important features of the formalisation of EXTRA are: the interpretation which, similarly to the inherited interpretation of types [2], takes into account type hierarchy; and the interpretation of so-called reference types, which correspond to classes in the terminology of IQL and ours.

By the presented formalization we define a data model which is related to the family of languages popularly called description logic (abbr. DL) [5]. Description logics evolved from KL-ONE [3]—a frame-based language that uses *concepts* and *roles* for describing the modeled domain. Besides the similarities in the structural properties of DL concepts and objects in our formalisation, they also share some common operations: for instance, the subsumption test [5] is related to testing the validity of the relationship $\preceq_v$ between values, and, computing Least Common Subsumer [7] corresponds to the model-based operations *lub-set* and *glb-set* [17].

7. Concluding Remarks

In this paper, we studied structural aspects of object model through the formalization which unifies the instance and schema levels of object model. It is shown that the structural part of object language can be seen as three partially ordered sets: partially ordered sets of identifiers, values and objects. The relationships between the elements of these sets are presented by defining the semantics of the main features of object model, such as classes, types, inheritance, and instantiation. The consequences of removing the boundary between the intensional and the extensional levels of object model are studied. It is shown that only a few changes and additional constructs need to be introduced in order to represent the formal view of the extended object model.

The presented object model has been implemented in the framework of the object algebra [16]. In brief, object algebra includes, in addition to the relational and nested-relational operations also operations for the manipulation of extensional and intensional parts of object repositories and the operations for the efficient manipulation of complex composite objects. Object algebra is implemented as a query language of the system for querying and integration of Internet data [18].

References

[1] S. Abiteboul, R. Hull, *IFO: A Formal Semantic Database Model*, ACM Trans. Database Systems, Vol.12, No.4, 1987

[2] S. Abiteboul, P.C. Kanellakis, *Object Identity as Query Language Primitive*, ACM SIGMOD 1988

[3] R.J. Brachman, J.G. Schmolze, *An Overview of the KL-ONE Knowledge Representation System*, Cognitive Science, Vol.9, No.2, 1985

[4] C. Beeri, *A Formal Approach to Object-Oriented Databases*, Data & Knowledge Engineering, No.5, 1990

[5] A. Borgida, *Description Logics in Data Management*, IEEE TKDE, Vol.7, No.5, October 1995

[6] A. Borgida, R.J. Brachman, D.L. McGuiness, L.A. Resnick, *CLASSICS: A Structural Data Model for Objects*, SIGMOD 1989

[7] W.W. Choen, A. Borgida, H. Hrish, *Computing Least Common Subsumers in Description Logics*, Proc. AAAI Conference, 1992

[8] L. Cardelli, *A Semantic of Multiple Inheritance*, Information and Computation, 76, 138-164, 1988

[9] R. Fikes, T. Kehler, *The Role of Frame-Based Representation in Reasoning*, Comm. of ACM, Vol.28, No.9, Sept. 1985

[10] A. Goldeberg, D. Robson, *Smalltalk-80: The Language and Its Implementation*, Addison-Wesley Publishing Company, 1983.

[11] James Gosling, Bill Joy, Guy Steele and Gilad Bracha, The Java Language Specification Third Edition, Addison Weseley.

[12] M. Hammer, D. McLeod, *Database Description with SDM: A Semantic Database Model*, ACM Trans. Database Syst. 6, 3 (1981), 351-386

[13] R. Harper, *Theoretical Foundations of Programming Languages (Draft)*, CMU, 2007.

[14] M. Kifer, G. Lausen, *F-Logic: A Higher-Order Language for Reasoning about Objects, Inheritance, and Scheme*, ACM SIGMOD 1989

[15] C. Lecluse, P. Richard, F. Velez, *O_2, an Object-Oriented Data Model*, ACM SIGMOD 1988

[16] I. Savnik, Z. Tari, T. Mohorič, 'QAL: A Query Algebra of Complex Objects', *Data & Knowledge Eng. Journal*, North-Holland, Vol.30, No.1, 1999, pp.57-94.

[17] I.Savnik and Z.Tari, *Querying Objects with Complex Static Structure*, Proc. of Int. Conf. on Flexible Query Answering Systems (FQAS'98), To appear, May 1998.

[18] I. Savnik, Z. Tari, 'QIOS: Querying and Integration of Internet Data', http://www.famnit.upr.si/~savnik/qios/, FAMNIT, April 2007.

[19] I. Savnik: On formalization of object model by unifying intensional and extensional representations, Technical Report, FAMNIT, April 2009.

[20] B. Stroustrup, *The C++ Programming Language*, Addison-Wesley, 3rd edition, 1995.

[21] B. Stroustrup, *Multiple inheritance for C++*, AT&T Bell Laboratories, The C/C++ Users Journal, May 1999.

[22] S.L. Vandenberg, *Algebras for Object-Oriented Query Languages*, Ph.D. thesis, Technical Report #1161, University of Wisconsin-Madison, July 1993 "

[23] G. Winskel, *The Formal Semantics of Programming Languages: An Introduction*, MIT Press, 1993 (1-8).