ISTITUTO DI ANALISI DEI SISTEMI ED INFORMATICA
CONSIGLIO NAZIONALE DELLE RICERCHE

A. Formica, M. Missikoff

REFINEMENT HIERARCHIES IN STRUCTURAL CONCEPTUAL MODELING

R. 528 Giugno 2000

Anna Formica and Michele Missikoff – Istituto di Analisi dei Sistemi ed Informatica del CNR, viale Manzoni 30 - 00185 Roma, Italy. Email: {formica, missikoff}@iasi.rm.cnr.it.

ISSN: 1128–3378

Collana dei Rapporti dell'Istituto di Analisi dei Sistemi ed Informatica, CNR

viale Manzoni 30, 00185 ROMA, Italy

tel. ++39-06-77161
fax ++39-06-7716461
email: iasi@iasi.rm.cnr.it
URL: http://www.iasi.rm.cnr.it

Abstract

Refinement hierarchies have been widely studied and applied in many different fields of *Computer science*. Seen from a conceptual modeling perspective, refinement essentially concerns the hierarchical organization of a set of concepts regarding entities of a given domain. Research on the modeling of the static aspects of entities, and their hierarchical organization, has produced a significant amount of results that can be found in the literature. However, very often it is not easy to read and compare results coming from different fields. There are different perspectives, the terminology greatly varies, the basic assumptions are different and the underlying semantics may hide discrepancies that are difficult to discover.

The intent of this paper is to provide a survey about a set of relevant proposals coming from three important fields of Computer Science: *Artificial intelligence* (and, in particular, *Knowledge representation*), *Programming languages*, and *Databases*. Furthermore, in the paper a reference frame is proposed, to be used for an homogeneous presentation of the different proposals, and a deeper understanding of refinement hierarchies.

1. Introduction

Refinement hierarchies represent a fundamental modeling mechanism in *Computer science*, widely studied and applied in many different fields. This popularity produced a great number of variations: *subtyping*, *inheritance*, *subsumption*, *subclassing*, *specialization* and *generalization*, *ISA*, and *AKO* (a kind of) are just a few terms, easy to find in the literature, closely related to the notion of refinement. With the advent of the *Object-oriented* paradigm [54], refinement hierarchies have received a new impulse: from programming languages to databases, from analysis and design to user interfaces, within the Object-oriented approach, refinement hierarchies are consolidating their popularity. Refinement hierarchies are part of the conceptual mechanisms that help human beings to organize their thoughts. They are constantly used (more or less consciously) by analysts and designers to observe the reality, study complex problems, and construct conceptual models of the problem domain, aiming at the development of computer applications.

Refinement hierarchies, seen from a conceptual modeling perspective, essentially concern the organization of a set of concepts according to a hierarchy. The intuition is straight forward: it is possible to assert that the term *person* denotes a more general concept than the ones denoted by *student* or *professor* and, conversely, *student* and *professor* represent refined concepts (specializations) with respect to *person*. There are many situations where things are not so straight forward. Therefore, it is useful to have a sound theory that operates on concepts and their definitions, allowing us to unambiguously determine if two concepts are in refinement relation. Deriving the refinement relation is the basic step necessary to build (and maintain) a correct refinement hierarchy. This issue has been referred to as *Taxonomic reasoning* in *Artificial intelligence*. The same issue, although in a different form, has been addressed in the *Programming languages* area, and referred to as *subtyping* (or, again with a different flavor, as *subclassing* or *inheritance*) [88]. For a complete treatment of the possible variations of inheritance in Programming languages, please refer to [81]. In particular, a user-defined type is a Programming languages notion closely related to what we referred to as a concept. Furthermore, subtyping is a binary relation closely related to refinement [28, 34]. In mathematical logic, a similar issue, referred to as *subsumption*, has been addressed and extensively studied [60]. Subsumption has been adopted by *Description logics* in Artificial intelligence, and by *Logic programming* to deal with hierarchical relations among assertions. In the *Database* area, after more than two decades of predominance of the relational model [83], the need for a more expressive data model led to the introduction of Object-orientation [53]. Also in this area there is a great number of proposals and it is not clear if one will prevail, but it is undoubtful that in the future of Database models there will be a significant amount of Object-orientation. Therefore, database designers [13] will be challenged by refinement hierarchies for a long while.

In building a conceptual model, a fundamental difference exists between static and dynamic aspects. Static aspects pertain to objects, their information structure and the legal states they can assume[1]. Dynamic aspects pertain to the behaviour of the objects, the processes they are involved in and the operations (i.e., methods, according to the Object-oriented jargon) they can actively perform. Both static and dynamic aspects have been extensively studied at a conceptual level, but in modeling the latter there is still less consensus than on the former.

In this paper we address the structural conceptual modeling, that refers to the static aspects.

[1] State transitions pertain to dynamic modeling, the definition of the set of legal states pertains to static modeling.

Research on the modeling of static aspects has produced a significant amount of results that can be found in the literature. However, very often it is not easy to read and compare results coming from different fields. There are different perspectives, the terminology greatly varies, the basic assumptions are different and the underlying semantics may hide discrepancies that are difficult to discover. Its systematic use can be traced back to the '60s, with the advent of *Semantic networks* [73]. Later, in the early '80s, the activities in the field of *Conceptual modeling* demonstrated the importance of such a modeling mechanism in significant areas of Computer science. In treating the matter, we followed the organization proposed in the seminal book "On Conceptual Modeling" [27]. Therefore, the paper addresses conceptual modeling, with a focus on structural refinement hierarchies, in the perspective of Artificial intelligence (more precisely in *Knowledge representation*), in Programming languages, and in Databases. For each of the above areas a couple of representative proposals have been identified: they have been analyzed and illustrated, to emphasize their contribution to structural conceptual modeling and refinement hierarchies.

The intent of this paper is to provide a reference frame, to be used for an homogeneous presentation of the different proposals, and a deeper understanding of refinement hierarchies. To this end, the paper starts with a preliminary section where the basic notions, that will be addressed in the subsequent sections, are briefly defined. The definitions are given with a selected terminology, in order to avoid any bias towards a field that may assume a prominent role with respect to the others. In trying to find a balance between intuitiveness and formality, the basic modeling notions are presented by using a notation originally proposed by the authors for Object-oriented Knowledge representation: TQL [62, 45]. That section, that also traces the layout of the successive sections, first introduces the notion of an entity, i.e., a static concept. Entities are defined by giving the (internal) information structure and their relationships with other entities. Once we have the mechanisms to define entities, we need criteria to determine if two entities are in refinement relation, to organize them according to a hierarchy. One of the main motivations for using conceptual hierarchies is the possibility of defining a complex scenario in an incremental fashion [87]. This last point introduces the inheritance mechanism in the incremental definition of refined entities, starting from more general (or abstract, if you wish) ones. Structural hierarchies are processed aiming at verifying formal properties (e.g., *consistency*), or making explicit the definition of subentities for which only the incremental specification has been given. Another kind of processing concerns the correct placement of a new entity in an existing hierarchy.

In order to give a comparative view of the different proposals, it is necessary to take into consideration, besides the notation, the associated semantics. However, this aspect has not been addressed with similar levels of elaboration in the different fields. Some proposals just disregard semantic issues, others give quick hints, and yet others are based on a rigorous formal setting. Therefore, in this paper we will address the issue only when required, and the treatment will be at an intuitive level.

The rest of the paper is organized as follows. The next section presents the basic notions of structural conceptual modeling according to an Object-oriented approach. Section 3 addresses the area of *Knowledge representation* where, in particular, *Description logics* and *Concept lattices* are illustrated. In Section 4, concerning *Programming languages*, the approaches to refinement hierarchies in *Object-oriented programming* and *Logic programming* are presented. In Section 5, *Object-oriented databases* are elaborated. In particular, we focus on an object database standard ($ODMG$), and a commercial product ($O2$). The last part of Section 5 presents a proposal of

an object database model that merges different proposals, placing particular emphasis on the correctness of inheritance hierarchies. The paper ends with the concluding section that presents a synoptic view of the different proposals.

2. A reference frame for structural modeling

In this section, a brief introduction to structural conceptual modeling will be presented. The basic notions concerning the definition of an entity, and its information structure, will be initially given. Then, we will introduce the notion of a *related entity set* (referred to as a *conceptual model*), its hierarchical organization, and the typical processes that are performed on such a hierarchy. In order to introduce the different modeling notions presented in this section, we decided not to use any of the surveyed solutions reported in the paper, to avoid giving it a prominent role. Therefore, as already mentioned, we adopted TQL (*Typing and Query Language*), an Object-oriented modeling language originally proposed by the authors [44, 45]. The layout of this section will be replicated in the next sections, in order to illustrate the different proposals in a uniform way.

2.1. Entity

An *entity* denotes a set of individuals having common characteristics. The set of individuals denoted by an entity is referred to as an *extension* or *interpretation* of the entity. As we will see in the next sections, in the literature entities (but also related notions) have been referred to with different terms, such as *types*, *concepts*, *classes*, *ψ-terms*. An entity is specified by an *entity expression*. An entity expression has a left hand side, that is the identifying *tag* of the entity (essentially, its label or name), and a right hand side, the *entity definition*. The latter describes the (information) *structure* of the entity which, intuitively, gathers the set of characteristics typical of that entity. Furthermore, entity definitions are propedeutical for entering and manipulating information. For instance, *person* and *dog* are meaningful entity tags. An *entity set*, gathering a set of related entities with their definitions, is typically referred to as a *conceptual model*. In the database field, it corresponds to a *database schema*. However, we will see that not all entity sets can be considered schemas: they must be compliant with specific rules. The set union of the elements in the extensions associated with an entity set forms an *Universe of Discourse* (*UoD*). An individual compliant with the characteristics of an entity is referred to as an *instance* of that entity.

Entity expressions may be hierarchically organized according to a *refinement* (or *specialization*) relationship. Such a relationship between entities corresponds to the set inclusion between their extensions. For instance, the *student* entity is a specialization of the *person* entity (and *person* is a *generalization* of *student*), since every individual denoted by *student* is also an individual denoted by *person*. Furthermore, an individual compliant with the characteristics of *student* is expected to be compliant with the characteristics of *person*.

Sometimes it is necessary to express that an entity is a specialization of two different (generally, hierarchically unrelated) entities. In TQL, a specialization hierarchy is expressed by means of the ISA constructor. For instance, to indicate that a *working student* (*emp_stud*) is simultaneously a specialization of a *student* and an *employee* we write[2]:

[2]The syntax of TQL is sufficiently intuitive that allows us to omit its formal syntax.

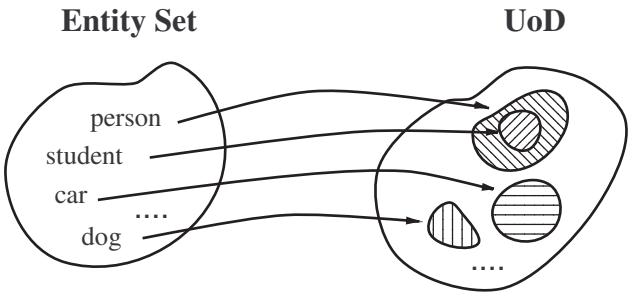

Figure 1: Entity set and UoD

`emp_stud` := ISA `student employee`

In the case of multiple specialization, as shown in the example, the resulting extension is the intersection of the extensions of the entities appearing in the ISA construct.

2.2. Entity expression

The definition of an entity consists of a set of characteristics typical of the individuals denoted by that entity. Characteristics can be structural, when pertaining to static aspects, or behavioral, if they describe dynamic aspects. In this paper we focus on the static aspects of entities and, therefore, entity definitions will be restricted accordingly. An entity definition provides criteria to establish if an individual is an instance of the corresponding entity (i.e., if it is compliant with the characteristics of that entity). The structure of an entity is defined by a set of *properties*, where each *property* can be seen as a variable devoted to represent a (static) phenomenon characteristic of the entity. For instance, the structure of the *person* entity may be defined by the properties *name*, *age*, and *pet*. An instance of an entity is created by associating a constant (or more, in the case of multivalued properties) with each property of the entity, and then by identifying such a structure with a label. Therefore, an instance is a *labeled set of labeled constants*. The identifying label of an instance is referred to as *instance tag* (or *object identifier*, *oid*, in the Object-oriented approach [55]). We distinguish the case in which the constant is an element of a basic domain, such as *string* and *integer*, and the case in which the constant is an instance tag, i.e., it is a reference to an element of an entity extension. The former will be referred to as *value* and the latter as *reference*. Checking an individual to identify the entity it is an instance of is often referred to as *instance checking* [39], or *classification* [75].

2.3. Domain restriction

The majority of languages for entity definitions have introduced mechanisms for the restriction of the domains from which properties can take values (*domain restriction*). This notion is not present in the first Object-oriented or logic languages, such as *Smalltalk* [90] or *Prolog* [33], respectively. Domain restriction is used in the entity definition to restrict the set of admissible constants for the properties. Such a mechanism is implemented by associating each property with a domain specification.

For instance, a domain restriction for the properties *name*, *age*, and *pet* of the above mentioned *person* entity, can be expressed by associating with them the domains *string*, *integer*, and *dog*, respectively. Notice that while the first two domains, *string*, and *integer*, are immutable and axiomatically given, the domain of *dog* depends on the *UoD*. (From a technical point of view, if the entity set corresponds to a database schema, the *UoD* is the *content* of the database, which evolves in time with update operations.)

The example below, written according to the syntax of the *TQL* language, shows a definition of the *person* entity introduced above:

$$\texttt{person} := [name : \{\texttt{string}\}+, age : \texttt{integer}, pet : \{\texttt{dog}\}]$$

where, in particular, it specifies that a *person* instance must have at least one *name*, that is a *string*, exactly one *age*, that is an *integer*, zero or more *pets*, that are instances of the *dog* entity.

In the case of multivalued properties (denoted by curly braces), it is also possible to express *cardinality constraints* that allow one to define the minimum and maximum number of values that can be associated with a given property. For instance:

$$\texttt{student} := [name : \{\texttt{string}\}, reads : \{\texttt{book}\}_{3,5}]$$

establishes the minimum and maximum number of books read by a *student*. Furthermore, the absence of curly braces, is a short form standing for minimum and maximum cardinality equal to one. *TQL* also allows interval domains to be expressed (e.g., subsets of the integer domain), that can be specified by enumeration or indicating the interval extremes. For instance, we can characterize a *teen_ager* by associating the interval domain (13..19) with the property *age*, as follows:

$$\texttt{teen\_ager} := [name : \texttt{string}, age : (\texttt{13..19}), pet : \texttt{dog}]$$

As already mentioned, an individual that is an instance of an entity is constructed by instantiating all the properties of that entity, and by labeling the set of labeled constants. For example, the following structure:

$$(\#p14\text{: } [name : \{Bob, Tom\}, age : 30, pet : \#d7])$$

represents an individual which is an instance of the *person* entity defined above. Such an instance is uniquely identified by the tag $\#p14$. In the above example we assume that, in the *UoD*, $\#d7$ is the identifying tag of a *dog* instance. Therefore, a property establishes a binary relation over the instances of the related entities (the definee and the definer entity that appear to the left and to the immediate right of the property, respectively).

Entity expressions may be recursive, i.e., their definitions directly or indirectly may refer to the entity tags being defined. For instance, below a self recursive entity expression is given:

$$\texttt{expert} := [name : \{\texttt{string}\}, skill : \texttt{string}, colleague : \{\texttt{expert}\}]$$

2.4. Refinement hierarchy

Refinement hierarchies have attracted much attention from the research areas of Artificial intelligence and Computer science, in particular, from Knowledge representation and Object-oriented programming. The first systematic work on abstraction hierarchies originated in Knowledge representation, within Semantic network models [22, 21]. Informally, an entity definition is a refinement of another one if the former has a definition (structure) that is more accurate than the latter. Accuracy can be achieved by adding properties (refinement by extension). Intuitively, we may say that a larger description is more accurate than a shorter one (of course, in absence of redundancy). Another mechanism of refinement can be obtained by a more accurate domain specification associated with a property (refinement by restriction). Refined entity definitions may be expressed in an *expanded* or a *compacted* form. In the former case, the definition of the refined entity has all the properties explicitly given, whereas a compacted form includes references to already defined entities. Compacted definitions require a specific construct that allows other entities to be referenced. The latter will be referred to as *superentities*, while the refined ones as *subentities*. It is evident that an entity, in an entity set, can be a superentity and a subentity at the same time. The use of compacted definitions allows more succint and readable entity sets to be defined.

The refinement mechanism has also an important semantic import: it is related to the inclusion among the sets of individuals the entities denote.

For instance, consider the entity *person*, as defined above, and suppose we want to introduce the entity *student* as a refinement of the former, by specifying for it the additional property *college*. By using an expanded definition, we may specify *student* as follows:

$$\texttt{student} := [\mathit{name} : \{\texttt{string}\}+, \mathit{age} : \texttt{integer}, \mathit{pet} : \{\texttt{dog}\}, \\ \mathit{college} : \texttt{string}]$$

whereas, by using a compacted definition, we may take the *person* definition of the previous section and add the new property by using the ISA construct:

$$\texttt{student} := \textbf{ISA}\ \texttt{person}\ [\mathit{college} : \texttt{string}]$$

As above mentioned, an entity may also be refined by restricting the domain of a property of a superentity. For instance, the *teen_ager* entity can be defined, in a compacted form, as a refinement of *person* having redefined the *age* property, as follows:

$$\texttt{teen\_ager} := \textbf{ISA}\ \texttt{person}\ [\mathit{age} : (\texttt{13..19})].$$

The mechanism of refining the domain associated with a property, already defined in a superentity, is also referred to as *overriding*. However, languages with a rich expressive power, e.g., which include multivalued properties, require more elaborated refinement mechanisms, that are not limited to the property set inclusion or domain restriction (e.g., set cardinality restriction).

As already seen in the previous subsection, refinement can be single or multiple, depending on the number of arguments in the ISA construct. The example given in Subsection 2.1 shows *emp_stud* as a case of multiple refinement. Also in this case, there exists an equivalent expanded definition. For instance, suppose that *student* is defined as in the example above, and *employee* is defined as:

$\texttt{employee} := [name : \{\texttt{string}\}+, company : \texttt{string}, salary : \texttt{integer}]$

Starting from a compacted definition, in order to have all the properties of a subentity explicitly given, a rewriting mechanism based on the union of properties is needed. The expanded form of *emp_stud* results to be:

$\texttt{emp\_stud} := [name : \{\texttt{string}\}+, age : \texttt{integer}, pet : \{\texttt{dog}\},$
$college : \texttt{string}, salary : \texttt{integer}, company : \texttt{string}]$

The rewriting mechanism used to transform a compacted definition into its equivalent expanded form is generally referred to as *inheritance*, as described below.

2.5. Hierarchy processing

In the previous section we introduced a rewriting mechanism for compacted definitions, that allow the expanded forms to be derived by inheriting the properties of the superentities. Such a mechanism consists, essentially, in the removal of the ISA construct, and the derivation of the structure of the subentity, by performing the union of the properties of the referred superentities (and, if present, the explicit component of the subentity). In the case of superentities having common properties, *inheritance conflicts* arise. Such conflicts are solved by determining the domain restriction to be associated with the properties derived for the subentity. In the case of overriding, the local properties, explicitly given, are assumed to replace inherited properties with the same name (but a compatibility check must be performed, as explained later).

Consistency is a key property of refinement hieararchies, strictly related to inheritance conflicts. It is defined as follows (rephrasing, with our terminology, a definition given in [94]):

"in a *consistent refinement hierarchy* each compactly defined
entity, when expanded, must be a refinement of each of its superentities".

Consistency is essentially related to inheritance conflict resolution, in particular it is violated in the presence of conflicting properties having domains with empty intersection.

For instance, in the case of the *emp_stud* entity defined above, the refinement hierarchy is consistent because the expanded definition derived for it is a refinement of both the *student* and *employee* superentities. In particular, in this case, the *name* property generates an inheritance conflict that can be trivially solved, being the basic domain associated with it, in the superentities, the same.

Conversely, there are cases where the conflict requires an extensive processing to be solved, or it appears evident that a solution does not exist. Consider the following entity set:

$\texttt{student} := [name : \texttt{string}, phone : \texttt{string}, college : \texttt{string}]$
$\texttt{worker} := [name : \texttt{string}, phone : \texttt{integer}, salary : \texttt{integer}]$
$\texttt{work\_stud} := \textbf{ISA}\ \texttt{student worker}$

In this case, an inconsistent refinement hierarchy has been defined. In fact, in the definition of *work_stud*, the *phone* property can be associated with either *integer* or *string*. But, in both cases, the resulting definition is not a refinement of both the superentities *student* and *worker*.

Furthermore, being the intersection of *integer* and *string* domains empty, a non-empty domain that is a restriction of both does not exist.

2.6. Semantics of an entity

As already mentioned, the set of individuals denoted by an entity is an interpretation, or extension, of that entity. Indeed, we are not interested in any interpretation, but only in the interpretations for which all the individuals are instances of (at least) one entity (i.e., they are compliant with an entity structure). Such interpretations are referred to as *semantic models* (models for short) of the entity. In this section we give a brief account of the research on entity semantics. (A more elaborated treatment is beyond the scope of this paper and can be found in the literature mentioned below.)

In the literature, according to the expressiveness of the language proposed, various kinds of semantics have been defined [35]. For instance, in the context of *Deductive databases* the *inflationary*, *stratified*, and *well founded* semantics have been defined to deal with negation and recursion [5]. In the context of *Object-oriented databases* there are various proposals based on denotational semantics, such as in [11], or based on an algebraic specification approach, such as in [17].

In this survey, since we avoid negation and we focus essentially on conjunctive, recursive structures, it is worth recalling, within the denotational semantics, the *least* and *greatest fixed point* semantics. Furthermore, it is worthwhile to consider the *descriptive* semantics [65, 66], which has been proposed in the context of Description logics. In our opinion, it is also significant in other contexts, and in particular in Object-oriented databases [15].

The three mentioned semantics, i.e., the least, the greatest fixed point, and the descriptive semantics, agree on non-recursive entities, while they differ in the case of recursive entity expressions. In particular, according to the least (greatest) fixed point semantics, only the smallest (greatest) model, among all possible models, is the intended meaning of the entity. Conversely, according to the descriptive semantics, all possible models are allowed. For instance, consider the following recursive entity:

$$\texttt{tennis\_player} := [name : \texttt{string}, friend : \texttt{tennis\_player}]$$

and suppose that the UoD contains two individuals, $\#o1$, $\#o2$, whose structures are defined, respectively, as follows:

$\#o1$: $\{name : John, friend : \#o1\}$
$\#o2$: $\{name : Mary, friend : \#o2\}$

Then consider the sets:

$\{\ \}$, $\{\#o1\}$, $\{\#o2\}$, $\{\#o1, \#o2\}$,

that are all the possible models of *tennis_player*. Indeed, according to the least and greatest fixed point semantics, the intended meaning for the *tennis_player* entity is the empty set (the smallest model) and the set $\{\#o1, \#o2\}$ (the greatest model), respectively. Whereas, according to the descriptive semantics, all the four sets are possible meanings of *tennis_player*.

With regard to hierarchies, semantic constraints related to the intersection of the extensions

of sibling entities (i.e., entities having a common superentity) are often proposed, allowing the notions of *disjoint* and *overlapping* refinement to be introduced. In particular, a refinement is *disjoint* or *overlapping* if the intersection of the models of the sibling entities is empty or not, respectively. (Notice that in the case of multiple refinement - i.e., entities having a common subentity - the extensions of the superentities are always assumed to be overlapping.) A formal treatment of overlapping and disjoint refinement has been investigated in [48], within the *Object-Role Modeling* (*ORM*) techniques.

3. Knowledge representation

Knowledge representation (*KR*) is one of the key fields of *Artificial intelligence* (*AI*) [24]. KR is per se a very rich and active field that, roughly speaking, can be divided according to two major areas: one grounded on logic [46], and the other one grounded on algebra and graph-theoretic foundations (e.g., *Semantic networks* [22] and, to mention a specific proposal, *Conceptual graphs* [78, 79]). In this paper, we selected *Description logics* (*DL*) (previously known as *Terminological logics* or *Concept languages*) [74, 39], and *Concept lattices* (*CL*) (also referred to as *Galois graphs*) [89], as representative proposals of these two areas, respectively, since they pay particular attention to conceptual hierarchies.

In both the subsections below, we start by introducing the notion of a *concept* that corresponds to the entity notion of the reference model presented in the previous section.

3.1. Description logics

DL have acquired particular relevance in the last years. "*DL* have been designed for the logical reconstruction and specification of Knowledge representation systems descending from *KL-ONE*, such as *Back*, *Classic*, $\mathcal{KRIS}$, and *LOOM*" [74]. In addressing concept hierarchies, the main issue of *DL* is *Taxonomic reasoning*, i.e., the determination of the right place of a concept in a taxonomy of concepts [19]. Taxonomic reasoning is based on the computation of the refinement relation between concepts, that in this field is referred to as *subsumption*.

3.1.1. Concept

In DL, entities are referred to as *concepts*. A concept therefore denotes a set of individuals, and is defined by means of a concept expression. In early proposals, concept expressions were *unamed*, that is without concept tags. In such a case, a concept expression coincides with a concept definition, and it is not possible to reference it by name. Indeed, even in this early form, concept names (tags) are present and are used to form concept expressions. However, they are not associated with a definition, and their semantics is axiomatically assumed: a concept name is a sort of atomic concept.

We recall that unamed concepts and concept subsumption have been originally investigated by Brachman and Levesque in the seminal paper [23]. Being the unamed concepts less expressive than the named ones (for instance, unamed concepts do not allow recursive definitions), research has later focused on named concept expressions.

Named concept expressions may be *primitive* or *defined* [49, 39]. A *primitive* concept expression has a definition that represents only necessary conditions, for an individual to be an instance of the concept. In this case, the concept definition denotes a superset of the instances denoted by the concept name. Intuitively, if:

person: has a *name* and an *age*

represents a primitive concept, then the set of the *persons* is a subset of all the possible individuals having a *name* and an *age* (that, for instance, includes dogs as well).

A *defined* concept is a concept expression where the definition represents necessary and sufficient conditions. Intuitively, if we want to transform the primitive *person* concept into a defined one, one solution (scarcely feasible, though) is to add a list of properties that, collectively, only pertain to persons. In general, since it is difficult to build a definition based on all the properties that uniquely characterize a specific entity, a defined concept is obtained by using another (typically primitive) concept. Therefore, we may write the above concept in a defined form:

person: a *human* which has a *name* and an *age*.

Therefore, assuming that *human* is a primitive concept that axiomatically denotes the set of all possible human beings, the above concept expression excludes dogs (or other entities having *name* and *age*) being part of the concept extent.

However, in some papers, see for instance [67], *primitive* concepts are simply concept names (i.e., atomic concepts) that are not associated with any definition, whereas $defined$ concepts are always named concepts, whose associated definitions denote necessary and sufficient conditions for their instances identification.

3.1.2. Concept expression

The expressions of primitive and defined concepts are characterized by specific constructors, connecting the concept name and its definition. In particular, defined concepts are specified by using the symbol "$\doteq$", whereas primitive concepts are expressed by the "$\sqsubseteq$" symbol.
In DL, the properties used in constructing a concept definition are referred to as *roles*.

Given a role R and a concept C, below the operations that allow primitive and defined concepts to be constructed are presented. Notice that, in this paper, we restricted such operations to the ones that are, somehow, common to all the DL languages found in the literature [49, 66].

- **Concept conjunction**: allows the specification of the superconcepts of the concept being defined (it is equivalent to the ISA construct). It is expressed by using the infix "$\sqcap$" operator. For instance:
  $work\_stud \doteq (employee \sqcap student)$
  is a concept whose name is *work_stud*, defined through the superconcepts *employee* and *student*. In particular, the *work_stud* individuals are exactly the ones denoted by the intersection between the extensions of *employee* and *student* concepts.

- **Value restriction**: this operation allows one to express the *universal role quantification* ($\forall R.C$), and the *existential role quantification* ($\exists R.C$). For instance:
  $educated\_person \doteq (\forall\ child.student)$
  $student \sqsubseteq (\exists\ friend.dog)$
  are two concepts, the former establishing that the *educated_persons* are exactly the individuals having all *children* that are *students* (or without offsprings), the latter stating that the *students* are individuals having at least one $friend$ that is a *dog*.

- **Number restriction**: since a role may be multivalued, the number restriction gives the possibility of specifying cardinality constraints. Notationally such a constraint is expressed by $(\geq n\ R)$ and $(\leq n\ R)$, where n is a natural. For instance:
  $person \sqsubseteq (\geq 2\ friend)$
  is a concept stating that each *person* has at least two $friends$ (notice that it is not required to specify the value restriction of a given role).

More complex concepts may be formed by composing the above constructs, as for instance:
educated_parent $\doteq$ (*adult* $\sqcap$ ($\forall$ *child.student*) $\sqcap$ ($\geq$ 2 *child*)
$\sqcap$ ($\exists$ *boss.manager*))
that describes an *educated_parent* as an *adult*, who has at least two *student children* and, furthermore, at least one *boss*, who is a *manager*.

Notice that the above operations have been expressed according to an abstract syntax widely accepted by the AI community [49]. However, in KR systems, such as *KL-ONE* [25], and the related *Classic*, *Back*, $\mathcal{KRIS}$ and *LOOM* systems [91], a more concrete syntax is used, recalled below by means of the following correspondence table:

abstract	vs	*concrete* syntax
$\sqcap$ (infix)		**and** (prefix)
$\forall$		**all**
$\exists$		**some**
$\geq$		**atleast**
$\leq$		**atmost**

According to the concrete syntax, the above example concerning an *educat ed_parent* has the form:

educated_parent $\doteq$ (**and** *adult* (**all** *child student*)
(**atleast** 2 *child*) (**some** *boss manager*))

3.1.3. Hierarchy processing

In DL, the refinement relationship between concepts is referred to as *subsumption*. Rather then deriving the expanded form of the superentities, in DL the main interest has been checking subsumption between concepts, in particular, between defined concepts. Subsumption is a semantic notion, defined as follows [23]:

a concept c is subsumed by a concept c' if and only if
any instance of c is also an instance of c'.

In the literature, various algorithms for checking subsumption have been introduced (see, for instance [23, 66]). Subsumption algorithms are required to be *sound* and *complete*. In particular, *soundness* implies that, given two defined concepts c and c', if the algorithm reports that c is subsumed by c', then any instance of c is also an instance of c'. *Completeness* implies that, if any instance of c is also an instance of c', then the algorithm reports subsumption.

For example, consider the following concepts, defined according to the concrete syntax recalled above:

employee $\doteq$ (**and** *worker* (**all** *project string*))
work_empl $\doteq$ (**and** *worker student* (**all** *project string*)
(**all** *friend person*))

These concepts are in subsumption being any superconcept and role (with the corresponding value restrictions) of the former present in the definition of the latter.

In the case of recursive concepts, the picture gets more complicated since a concept expression does not correspond to a finite parse tree and, therefore, a direct comparison of the two structures is not feasible. Furthermore, as anticipated, the semantics of recursion is not uniquely agreed upon and the subsumption relationship may hold or not, depending on the chosen semantics. Consider the following example:

$student \doteq$ (**and** (**all** $name$ $string$) (**atleast** 1 $name$)
(**all** $friend$ $student$) (**atleast** 1 $friend$))
$employee \doteq$ (**and** (**all** $name$ $string$) (**atleast** 1 $name$)
(**all** $friend$ $employee$) (**atleast** 1 $friend$))

According to the greatest fixed point semantics, the models of these concepts always coincide, therefore subsumption between them holds, trivially. Whereas, according to the descriptive semantics, these concepts are not in subsumption, because there exist models of *student* and *employee* for which the set inclusion does not hold (see Subsection 2.6) [66].

One of the main challenges of the analysis performed in the field of DL is the identification of fragments of formal logic that allow efficient Taxonomic reasoning methods to be defined. Since the more expressive the language, the harder the reasoning, the challenge consists in determining a trade-off between the expressivity of the representation language and the possibility of reasoning with the expressions represented with such a language. Computational complexity results about subsumption show that in the case of restricted languages including the **and**, **all**, and **some** costructors and unamed concepts, subsumption can be computed in polynomial time [23]. However, in the case of languages containing the **and**, and **all** constructors, with named and unamed concepts, subsumption becomes intractable, and in particular it is co-NP-complete [67, 38], or even undecidable for full $KL\text{-}ONE$ concepts [77].

3.2. Concept lattices

CL have been introduced aiming at the organization of the instances that are present in a given domain, by grouping them in a coherent fashion. This is obtained by first identifying the (relevant) objects of the observed domain, together with their (relevant) features. In this perspective, an entity (concept) is not an abstraction but, on the basis of the observation of the reality, it is a clustering of objects having common features (attributes). In particular, CL, also referred to as *Galois graphs*, have been introduced by Wille within the *Formal Concept Analysis* [89]. Formal Concept Analysis provides a conceptual framework for the analysis and visualization of data, in order to make them more understandable. It is based on *lattice theory* [18], a well established mathematical discipline that has been applied in many different realms, like Psichology, Sociology, Medicine, Linguistics, and Computer science. In one of his first works, Birkoff states that "lattice theory provides a proper vocabulary for discussing order, and especially systems which are in any sense hierarchies" [20].
CL were originally proposed for conceptual classification and data analysis in AI, within the KR field [37]. More recently they revealed very interesting in providing a basis for the construction of inheritance hierarchies in Object-oriented databases [92], [63].

The approach adopted in Formal Concept Analysis is somehow more extensional with respect

to the ones illustrated till now. In fact, the development of such a theory starts from the definition of a set O of *objects* (individuals), a set A of *attributes* (property labels), that are considered relevant to a given application domain, and a binary relation R among them. Such a triple of sets (O,A,R), i.e., the collections of objects, attributes, and the binary relation among them, is referred to as a *context*. Starting from a context, a set of possible *concepts* can be derived, each of which is defined by clustering the sets of objects that own the same sets of attributes, within the given context. All possible concepts that can be derived from a context can be organized according to a lattice, referred to as a *concept lattice*.

In Formal Concept Analysis, the notion of a concept is very similar to the notion of an entity as defined in Section 2.[3] However, with respect to the proposals illustrated till now, in CL there are neither concept labels, nor declared associations of concepts by means of attributes (i.e., relationships). A concept is derived from the context, on the basis of the existing relations among objects and attributes. It is completely specified by its sets of objects and attributes, referred to as *extent* and *intent* of the concept, respectively. The extent of a concept corresponds to the set of individuals denoted by the concept, whereas the intent corresponds to the concept expression (that corresponds to the entity expression, as defined in Subsection 2.2).
Concepts are hierarchically related by means of a *subconcept-superconcept* relation. In particular, the extent of the subconcept is contained by the extent of the superconcept, whereas the intent of the superconcept is contained by the intent of the subconcept. These issues are better illustrated below.

3.2.1. Concept

As already mentioned, in Formal Concept Analysis a *concept* is defined within a *context*. A context is a triple (O,A,R), where O and A are two sets of elements called *objects* and *attributes*, respectively, and R is a binary relation between O and A. In particular, if oRa, for any $o \in O$ and $a \in A$, then we say that "the object o has the attribute a" or "the attribute a applies to the object o".

Given two sets E, I, such that $E \subseteq O$ and $I \subseteq A$, consider the *dual* sets E' and I', i.e., the sets defined by the attributes applying to all the the objects belonging to E and the objects having all the attributes belonging to I, respectively, that is:

$$E' = \{a \in A \mid oRa \ \forall \ o \in E\}$$
$$I' = \{o \in O \mid oRa \ \forall \ a \in I\}$$

Then, a *concept* of the context (O,A,R) is a pair (E,I) such that $E \subseteq O$, $I \subseteq A$ and the following conditions hold:

$$E' = I,\ I' = E.$$

The sets E and I are referred to as the *extent* and the *intent* of the concept, respectively. Therefore, a concept is a pair of sets where the former consists of precisely those objects which have all attributes from the latter and, conversely, the latter consists of precisely those attributes that apply to all objects from the former.

[3] Notice that, in Concept lattices, the term entity is used as a synonym of object, rather than of concept. We are aware that this terminological clash may reduce the clarity of the presentation, therefore we used the term object instead.

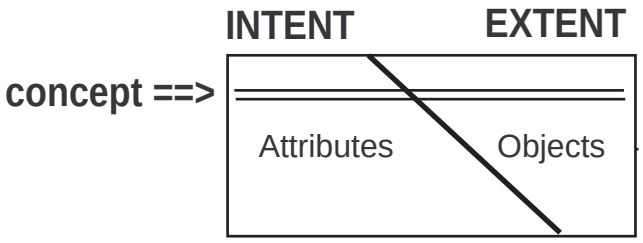

Figure 2: Concept extent/intent relationships

For instance, consider a context where:

O = {#b, #mb, #e, #s, #w},
A = {name, vehicle, college, salary, maker, color, power}

and R is specified by the Table 1.

	name	vehicle	college	salary	maker	color	power
#b					x	x	
#mb					x	x	x
#e	x	x		x			
#s	x	x	x				
#w	x	x	x	x			

Table 1: A Context

In this context, five objects are present, each of which identified by the "#" prefix symbol, and seven attributes. Notice that, as already mentioned, in a concept lattice entity tags, such as *student* or *bike*, are not expressed. Furthermore, there is no possibility of specifying concept relationships such as, for instance, *bike* associated with *student* by means of the *vehicle* property. Finally, predefined domains, such as *string* or *integer*, are not considered.

A concept of this context is, for instance, the pair:

({#e,#w}, {name, vehicle, salary})

since both the objects $\#e, \#w$ have at least the *name*, *vehicle*, and *salary* attributes, and no other objects of the context has all these three attributes. If we construct a table of concepts (i.e., a set of objects-attributes pairs), placing the more general ones in the top row, we obtain a pattern sketchly shown in Figure 2.

Notice that, given a context (O,A,R) and two concepts (E_1,I_1) and (E_2,I_2), the following conditions hold:

if $E_1 \subseteq E_2$ then $E_2' \subseteq E_1'$, for E_1, $E_2 \subseteq O$
if $I_1 \subseteq I_2$ then $I_2' \subseteq I_1'$, for I_1, $I_2 \subseteq A$,

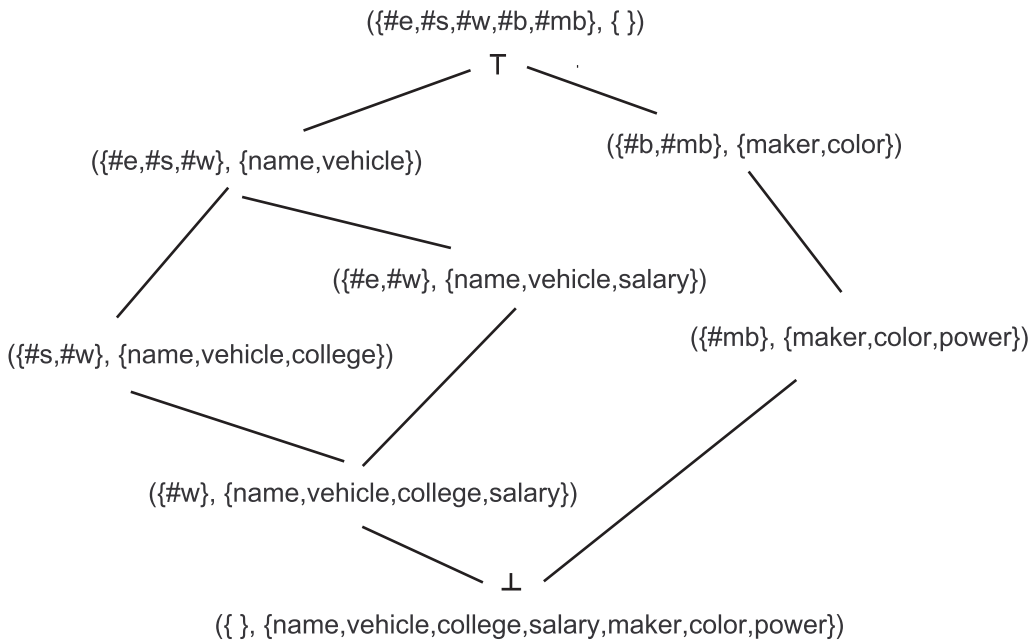

Figure 3: Concept lattice of the context of Table 1

that is, duality implies the opposite set inclusion in the case of both sets of objects and attributes. This situation is intuitively represented in Figure 2, where we can say that by adding attributes to a concept, the cardinality of its extent decreases or, vice versa, the reduction of the cardinality of the set of objects of a concept can be obtained by increasing the related intent cardinality (i.e., by identifying additional discriminating attributes).

3.2.2. Concept lattice

Given two concepts (E_1,I_1), (E_2,I_2) of a context (O,A,R), it is possible to establish an inheritance relation $(\leq)$ between them according to the following condition:

$$(E_1,I_1) \leq (E_2,I_2) \text{ iff } E_1 \subseteq E_2 \text{ (iff } I_2 \subseteq I_1).$$

In particular, (E_1,I_1) is called *subconcept* of (E_2,I_2) and (E_2,I_2) is called *superconcept* of (E_1,I_1).

Given a context (O,A,R), consider the set of all concepts of this context, indicated as $\mathcal{L}(O,A,R)$. Then:

$$(\mathcal{L}(O,A,R),\leq)$$

is a complete lattice called *concept lattice* (also referred to as *Galois graph*), i.e., for each subset of concepts, the greatest lower bound and the lowest upper bound exist [89]. (Notice that for lattices over sets with finite cardinality, the notions of complete lattice and lattice coincide [18].)

In the Concept lattice, nodes are labeled with the concepts of the context, and arcs are established among the nodes whose associated concepts are in $\leq$ relation. A node is associated

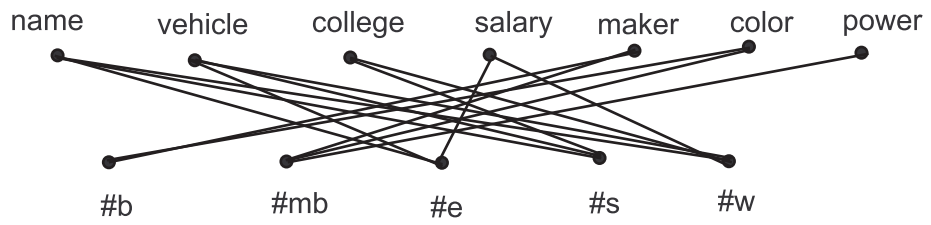

Figure 4: Bipartite graph representing the context of Table 1

with two components: the extent, that contains all the objects having the attributes specified in the second component, and the intent, that contains all the attributes applying to the objects specified in the first component. By the definition, the $\leq$ relation is a partial order relation that expresses a double inclusion among node components. In particular, given a node, say n: (i) the extent of n is contained in the extent of each of the ancestors of n, and (ii) the intent of n contains the intent of each of the ancestors of n.

The concept lattice has also two special nodes, the maximum and minimum nodes (labeled with $\top$ and $\bot$, respectively). The maximum and the minimum group all the objects and the attributes of the context, respectively. For instance, consider the context of Table 1. The concept lattice that can be derived from it is shown in Figure 3, and contains eight concepts.

Notice that in Figure 3, given two concepts, their greatest lower bound is the concept whose: (i) extent is the intersection of the extents of the two concepts, and (ii) intent is the union of the intents of the two concepts. On the other hand, their lowest upper bound is the concept whose: (i) extent is the union of the extents of the two concepts, and (ii) intent is the intersection of the intents of the two concepts. It is important to note that the lattice organization has been carefully conceived, but it is not the only possible. For instance, the same context can also be represented by using a bipartite graph, as shown in Figure 4, loosing readability and information about the hierarchical organization of concepts.

3.2.3. Hierarchy processing

In this area, the main activity related to hierarchy processing consists in computing the Concept lattice corresponding to a given context. In addition, the representation of a Concept lattice can be optimized, by defining the related *inheritance graph*. In fact, consider the example shown in Figure 3. By following the paths of the lattice it is easy to see that it contains redundant information concerning both attributes and objects. In particular, each node contains all the objects belonging to the extents of its descendants and all the attributes belonging to the intents of its ancestors. Now, by assuming that, for each node, attributes are inherited from the top ($\top$) and objects are inherited from the bottom ($\bot$), the Concept lattice of Figure 3 can be transformed into the inheritance graph of Figure 5. In this representation each node contains only the additional elements, objects and attributes, with respect to its descendants and ancestors. For instance, the node labeled with the concept $(\{\#e\}, \{salary\})$, inherits the object $\#w$ from the bottom, and the attributes *name* and *vehicle* from the top.

Another interesting processing activity performed in this research area is the derivation of all the *attribute implications* of a context. An attribute implication of a context (O,A,R) is a pair of subsets of A, say X, Y, denoted by "$X \rightarrow Y$", for which the following condition holds:

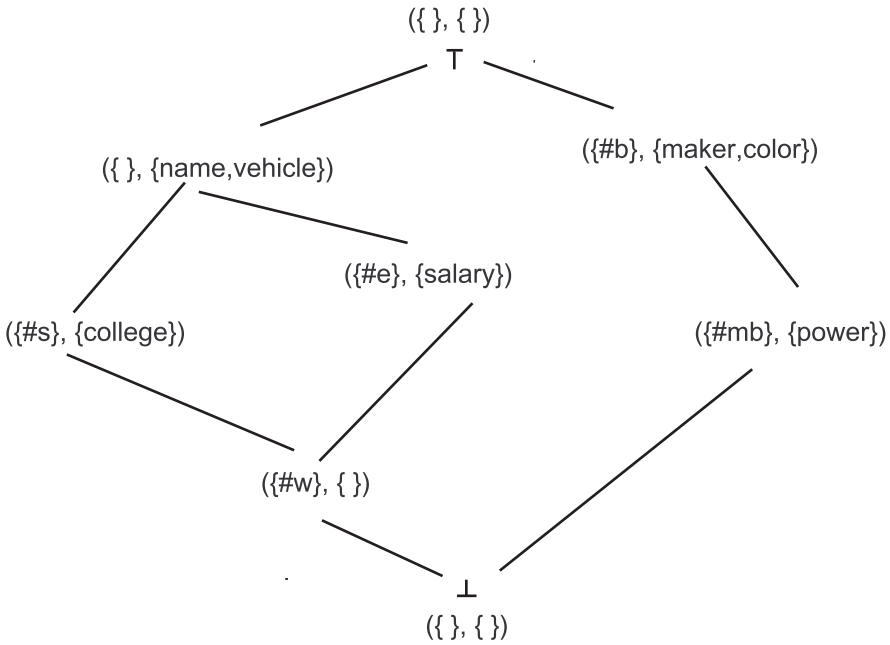

Figure 5: Inheritance graph of the context of Table 1

$$X' \subseteq Y'$$

that is, each object from O having all attributes of X has also all attributes of Y. This notion corresponds to attribute inheritance in Semantic networks.

For instance, in our example:

$$\{\text{college, salary}\} \rightarrow \{\text{name, vehicle}\}$$

is an attribute implication since every object (in this case only $\#w$) having the attributes *college* and *salary* has also the attributes *name* and *vehicle*. Similarly, also:

$$\{\text{maker, salary}\} \rightarrow A$$

is an attribute implication (where A is the set of all attributes of the context), since there are no objects in the context having the attributes *maker* and *salary*.

4. Programming languages

Refinement hierarchies have been proposed in *Programming languages* (PL) long ago. With the advent of *Object-oriented programming* (OOP) languages, such as *Smalltalk* [90], *C++* [40], *Eiffel* [61] and, more recently, *Java* [52], a programming style based on such hierarchies had a significant spread. However, in the mentioned OOP languages, refinement hierarchies did not have a sound theoretical basis. It is basically due to the work of Cardelli [1, 28, 36] if, in the

80's, OOP acquired a sound formal type theory. In the first part of this section, the key aspects of the OOP theory due to Cardelli will be illustrated, with particular attention to subtyping and type hierarchy.

In the area of *Logic programming* (LP), the need for more structured paradigms was emerging in the same period. For example, there have been several proposals to extend *Prolog* [33] with Object-oriented features (see $Prolog++$ [64]). Also in this case, the first proposals were mainly constructed with a pragmatic approach. A significant proposal that extends LP with Object-oriented features, and is based on sound theoretical foundations, is due to Ait-Kaci [6, 7]. This proposal is illustrated in the second part of the section.

4.1. Object-oriented programming

As already mentioned, here we will present the work of Cardelli [28], selected for the central role that it plays within the OOP community (and beyond). The presentation will follow the usual organization, addressing first the definitional notions and then focusing on hierarchy processing of the structural aspects. The rich treatment of this proposal concerning the functional aspects will not be addressed since it is beyond the scope of this paper.

4.1.1. Types and type expressions

In OOP, entities are referred to as *record types* (*types* for short). A type has a tag and a definition. A tag uniquely identifies a definition. A definition provides the necessary and sufficient conditions for an individual to be an instance of that type. Therefore, every type tag stands for the corresponding definition and does not carry any semantics per se. This approach is also referred to as the *structural approach*.

In a type, the properties are referred to as *fields* or *properties*. In a property, domain restriction is performed by associating each property name with a type definition or a type tag, either axiomatically given, such as *string* or *integer* (also referred to as *basic types*), or user-defined. In the latter case, the tag has to be properly defined. A property with an associated type tag or type definition will be referred to as a *typed property* (note that in OOP there are languages, such as *Smalltalk*, that do not require property to be typed). A user-defined type can be expressed in an expanded form, by explicitly listing all its typed properties, or in a compacted form, by using the (infix) **and** costructor. The arguments of the **and** costructor are referred to as *supertypes*. Properties are assumed to be singlevalued. Cardinality constraints are not addressed.

Example 4.1. In order to provide an example somehow related to the one illustrated in Subsection 3.2, consider the following set of types expressed according to the syntax of [28]:

$\texttt{student} = (\mathit{name} : \texttt{string}, \mathit{vehicle} : \texttt{bike}, \mathit{college} : \texttt{string})$
$\texttt{employee} = (\mathit{name} : \texttt{string}, \mathit{vehicle} : \texttt{motor\_bike}, \mathit{salary} : \texttt{integer})$
$\texttt{bike} = (\mathit{maker} : \texttt{string}, \mathit{color} : \texttt{string})$
$\texttt{motor\_bike} = (\mathit{maker} : \texttt{string}, \mathit{color} : \texttt{string}, \mathit{power} : \texttt{string})$
$\texttt{work\_stud} = \texttt{student}$ **and** $\texttt{employee}$

In the example, the first type expression has *student* as tag and *name*, *vehicle*, and *college* as properties. In particular, *vehicle* is typed with the user-defined tag *bike*, whose type definition

is given in the type set. □

Furthermore, we have recursive types, such as for instance:

$\texttt{person} = (name : \texttt{string}, vehicle : \texttt{car})$
$\texttt{car} = (name : \texttt{string}, owner : \texttt{person})$
$\texttt{student} = (name : \texttt{string}, friend : \texttt{student})$

where, in particular, *student* is a self-recursive type.

4.1.2. Subtyping

In OOP, the specialization relationship between types is referred to as *subtyping*. Subtyping is defined according to the following (recursive) syntactic rules:

- *every type is a subtype of itself*
- *a type t is a subtype of a type t' if t has all the properties of t', and possibly more, and the types associated with the common properties of t and t' are in subtyping relation.*

Notice that subtyping can be checked only for types that are in the expanded form, i.e., that do not contain the **and** construct. For instance, according to their explicit sets of properties, the *motor_bike* type defined above is a subtype of *bike*. As anticipated in Section 2, a pure structural approach, as the one adopted here for subtyping, may be misleading. Consider the following types:

$\texttt{person} = (name : \texttt{string}, age : \texttt{integer})$
$\texttt{dog} = (name : \texttt{string}, age : \texttt{integer}, breed : \texttt{string}).$

In this case, according to the above subtyping rule, *dog* is a subtype of *person* since, as already mentioned, their tags do not carry any semantics and just stand for their definitions.

With regard to recursive types, the subtyping rule has been deeply investigated in [2]. It can be summarized as follows:

t is a subtype of t' if, by assuming that subtyping between the type tags t and t' holds, it is possible to prove that subtyping between the type definitions of t and t' holds.

Therefore, it is easy to see that, if *person* and *dog* are two recursive types defined as follows:

$\texttt{person} = (name : \texttt{string}, friend : \texttt{person})$
$\texttt{dog} = (name : \texttt{string}, friend : \texttt{dog}, breed : \texttt{string})$

dog is a subtype of *person*.

The semantic import of the subtyping relationship is established by the following theorem (*Semantic Subtyping*) (see [28]):

if t is a subtype of t' then the set of instances of t
is contained in the set of instances of t'.

For instance, in the Example 4.1, any instance of *motor_bike* is also an instance of *bike*. In fact, any instance of *motor_bike* has a structure that contains a substructure (i.e., the properties *maker* and *color*) that coincides with the structure of *bike*. Here we are addressing the semantic issues at an intuitive level. On a more formal ground, it is important to point out that in OOP the greatest fixed point semantics has been adopted [2]. Therefore, according to the last example above, the set of instances of *dog* is contained in the set of instances of *person* (as expected according to the Semantic Subtyping theorem mentioned above).

4.1.3. Type inference

Subtyping has been introduced in OOP mainly for *type checking* purposes. Type checking is a sort of protection mechanism for the functions, aimed at guaranteeing that a function gets input parameters, at run time, of the expected type. Type checking can be performed dynamically, at run time, for every function invocation, or statically, at compile time. The former requires additional code to be generated and, furthermore, it overloads program execution. Therefore, run time checking is not very popular, in favor to the former. Static type checking is generally preferred, since it represents a burden for the compiler only and aims at increasing the quality of the software.

Static type checking requires a type inference mechanism. The latter, given a symbolic expression of the language, aims at deriving a type for it, according to a set of *type inference rules*. In the case that no type can be derived, the expression is said to contain *type errors*. A full treatment of type inference theory is beyond the scope of this paper. Here we focus on type inference for *ground* expressions that are: (i) printable values (such as strings, integers,...), or (ii) instantiated structures, i.e., finite associations of printable values or, recursively, of ground expressions with properties. Such structures are also referred to as (instantiated) *record expressions* (*r_expressions*) and will be specified according to the notation of [28]. For instance:

($name$ = Bob, age = 30, $dept$ = math)

is an *r_expression* of the type:

$\texttt{student} = (name : \texttt{string}, age : \texttt{integer}, dept : \texttt{string})$

A set of type inference rules is also referred to as an *inference system*.

The basic rule of the inference system proposed by Cardelli is related to subtyping and can be summarised by the following theorem (*Syntactic Subtyping*):

if an r_expression has a type t, and t is a subtype of t',
then the r_expression has also type t'.

The inference system contains a rule for each specific syntactic construct. The type inference rule concerning r_expressions is the following:

if e_i *is a r_expression of type* t_i*, for* $i = 1 \ldots n$*,*
then the r_expression $(a_1 = e_1, \ldots, a_n = e_n)$
is of type $(a_1 : t_1, \ldots, a_n : t_n)$.

For instance, consider the Example 4.1, and the r_expression:

(*maker*=Fiat, *color*=red, *power*=1300hp) (*)

Then, according to the type inference rule for record types, it follows that this expression is of type *motor_bike*. Furthermore, according to the basic rule related to subtyping, we derive that this expression is also of type *bike*.

An inference system is required to be *sound* with respect to the semantics of the language (*Semantic Soundness*), that is:

if it is possible to derive that an r_expression e has a type t,
then the value denoted by e belongs to the domain denoted by t.

In general, many types can be derived for a given r_expression that, subject to the soundness of the inference system, are all in subtyping relation. For instance, for the r_expression (*) two types are derived, *bike* and *motor_bike*, that are in subtyping relationship.

4.1.4. Type checking

Type checking, that represents one of the key topics in the OOP field, aims at determining the coherence of the parameters that will be passed to functions at run time. Intuitively, the goal is to prevent that, in running a program, a function invocation causes an error due to actual parameters of a type incompatible with the one expected (e.g., trying to multiply two strings).

Static type checking algorithms are rather complicated, since they analyze the source code in order to follow the data flow, keeping track of the type that, step after step, each variable assumes. In general, many types can be derived for an r_expression according to a given set of type inference rules. In any case, if the inference system is sound, such types are in subtyping relationship among them. In order to choose any of the admissible types, a type checking algorithm must be defined. Therefore, in general, given any expression of an OOP language, the type checking algorithm operates in connection with a type inference system and aims at deriving a type for that expression which is compatible with the type required by the function. A type checking algorithm is required to be *sound* with respect to the inference system (*Syntactic Soundness*), that is:

if the algorithm succeeds and returns a type for an expression,
then it is possible to prove, within the type inference system,
that the expression has that type.

The static type checking algorithm for r_expressions proposed in [28] is specified by means of the type checking function $\mathcal{J}$, defined as follows:

$$\mathcal{J}[(a_1 = e_1, \ldots, a_n = e_n)]\mu = [(a_1 : \mathcal{J}[e_1]\mu, \ldots, a_n : \mathcal{J}[e_n]\mu)]$$

where μ is the type environment for variables. In particular, the type chosen for an expression is the most specialized one, according to the subtyping relationship ($\leq$):

$\mathcal{J}[(e\ has\ a\ type\ s)]\mu = t$, if $\mathcal{J}[e]\mu = t$ and $t \leq s$, else $fail$. (**)

The type checking algorithm has been proved to be syntactically sound with respect to the inference system (see definition above). As a result, from the *Semantic Soundness* seen in the previous paragraph, and the *Syntactic Soundness* we have:

if an expression can be successfully typechecked,
then it cannot produce run-time type errors.

For instance, it is easy to verify that the type checking function $\mathcal{J}$ returns the *motor_bike* type for the expression (*) (being subtyping reflexive, the (**) holds trivially). Notice that, as already shown in the previous paragraph, it is possible to derive that such an expression is of type *motor_bike* according to the inference system too (as expected, since the algorithm is sound with respect to the inference system).

Notice that being record types the focus of this paper, a simplified version of the original elaboration has been presented. Indeed, the import of the (**) is much wider and can be better appreciated when considering the type checking of complex expressions, including other constructs of an OOP language.

4.1.5. Hierarchy processing

In OOP, the refinement hierarchy processing, i.e., the rewriting mechanism that allows the derivation of the properties of subtypes (defined in a compacted form), is referred to as *normalization process*. The **and** construct is interpreted as a *meet* operation between types ($\downarrow$), being the resulting type the greatest common lower bound of the supertypes, according to the *subtyping* relation. In particular, the resulting type will have (at least) all the typed properties of the supertypes. In the presence of common properties, inheritance conflicts are solved by recursively applying the normalization process to the conflicting types.

If t, t_i, s_j, t'_i, v_k are type tags, and $(a_i : t_i, b_j : s_j)$, and $(a_i : t'_i, c_k : v_k)$ are two record types, where $i = 1 \ldots n$, $j = 1 \ldots m$, and $k = 1 \ldots r$, the normalization process can be formally defined as:

- $t \downarrow t = t$
- $(a_i : t_i, b_j : s_j) \downarrow (a_i : t'_i, c_k : v_k) = (a_i : t_i \downarrow t'_i, b_j : s_j, c_k : v_k)$

For instance, in the case of the Example 4.1, the expanded form of *work_stud* is:

$\texttt{work\_stud} = (name : \texttt{string}, salary : \texttt{integer}, college : \texttt{string},$
$vehicle : (maker : \texttt{string}, color : \texttt{string}, power : \texttt{string}))$

where the inheritance conflict *vehicle* has been solved by applying the normalization process to the conflicting types *bike* and *motor_bike*. According to the structural approach, the above type is equivalent to:

$\texttt{work\_stud} = (\mathit{name} : \texttt{string}, \mathit{salary} : \texttt{integer}, \mathit{college} : \texttt{string},$
$\mathit{vehicle} : \texttt{motor\_bike})$

since, as already mentioned, a type tag stands for its definition. Notice that the refinement hierarchy of this example is consistent (see Section 2), being *work_stud* a subtype of both its supertypes *student* and *employee*.

The normalization process performed according to the pure structural approach does not allow to deal with certain recursive patterns. For instance, consider the types:

$\texttt{employee} = (\mathit{name} : \texttt{string}, \mathit{friend} : \texttt{employee})$
$\texttt{student} = (\mathit{name} : \texttt{string}, \mathit{friend} : \texttt{student})$
$\texttt{work\_stud} = \texttt{student}$ **and** $\texttt{employee}$

In this case the inheritance conflict due to the $friend$ property cannot be solved by recursively applying the normalization process, since it would lead to an endless computation. Such a problem has been extensively investigated in [43], where an algorithm has been proposed that allows a set of types, for which the refinement process does not terminate, to be identified in polynomial time.

4.2. Logic programming

In LP there are different proposals, as for instance *Quixote* [93], *DOT* [84], *F-Logic* [56], *Feature Structures* [29], *Life* [6], that go beyond the more traditional languages (such as the wide spread *Prolog*). These languages introduce advanced formalisms to represent the structural aspects of an entity and the possibility of defining hierarchical relations among them. Among the cited languages, we selected Life as the representative one.

A logic program represents a formal theory, where both entities and individuals can be represented. This mixing of intensional and extensional knowledge is one of the main characteristics of the LP approach.

In terms of hierarchy, the main peculiarity of the Life approach is the clear separation between the entity expressions, that are not explicitly hierarchically related, and the entity names, that are formally organized in a hierarchy, referred to as *Signature for inheritance*. Therefore, a hierarchy of entity expressions is defined through the integration of the entity expressions and the Signature for inheritance. For the specific role given to entity names (and to distinguish this proposal from the previous one, referred to as structural approach) we will refer to Life as having the *naming* approach. Another characteristic of Life is the possibility of specifying *coreference constraints* in an entity expression. Coreference indicates that two different expressions share a common substructure. Notice that, in the KR and OOP fields, coreference constraints cannot be expressed in the structural part (i.e., in the entity expression) but require a language extension, concerning explicit integrity constraints.

In Life, entity expressions are referred to as ψ-terms; hierarchies can be expressed only among entity names (not expressions); hierarchy processing is performed according to a sort of $unification$ mechanism. The rest of this subsection will elaborate on these issues.

4.2.1. ψ-term

In Life, a partially ordered set of *type constructor symbols*, also referred to as *sort symbols* (*sorts*), is assumed to be given. A sort can be the label of a predefined domain, such as *string*, a user-defined label, such as *person*, or an individual, such as *mycar*. Such set of sorts is used to define the entity expressions that, in Life, are referred to as ψ-terms. A ψ-term can be *tagged* or *untagged*. A tagged ψ-term is either a variable (also referred to as *coreference tag*) or an expression of the form $X{:}t$, where X is the root variable and t is an untagged ψ-term. An untagged ψ-term is either *atomic* or *attributed*. An atomic ψ-term is a *sort* symbol, whereas an attributed ψ-term is, essentially, an entity expression, as described below.

4.2.2. Attributed ψ-term

An attributed ψ-term is an expression associating a root sort symbol with a finite set of properties. Each property is formed by an *attribute label* associated with a ψ-term through the "=>" symbol (domain restriction). Coreference constraints can be expressed within an attributed ψ-term. They are specified by using variables in uppercase letters.

For instance, according to the syntax of [6], the following is the *person* attributed ψ-term:

$$\texttt{person}(id => name(first => string; last => X : string);\\ father => person(id => name(last => X))$$

having *person* as root sort symbol (also referred to as *principal* type), and two attribute labels, *id* and $father$, each of which used to associate the root symbol *person* with a nested ψ-term. For instance, the ψ-term associated with the *id* attribute contains two sorts, i.e., *name* (root sort of the nested ψ-term) and *string*, and the attribute labels $first$ and *last*. The variable X indicates a coreference constraint, that is the last name of a person must be the same of his/her father. Of course, coreference constraints require the associated types to be the same (in this case *string*, that can be omitted in the latter case).

As already mentioned, individuals are considered sorts too. This is the reason why in Life we can also find attributed ψ-terms whose structures contain individuals. For instance, we could describe the entity whose father is a specific individual, say $p1$, as follows:

$$\texttt{person}(id => name(first => string; last => X : string);\\ father => p1(id => name(last => X))$$

However, in this paper, for sake of simplicity, we will focus on untagged ψ-terms (therefore without coreference constraints), since tagging does not play any specific role in hierarchical processing.

Consider the entity expressions of the Example 4.1 that, according to the ψ-term approach can be written as:

$$\texttt{student}(name => string; vehicle => bike; college => string)\\ \texttt{employee}(name => string; vehicle => motor\_bike; salary => integer)\\ \texttt{bike}(maker => string; color => string)\\ \texttt{motor\_bike}(maker => string; color => string; power => string).$$

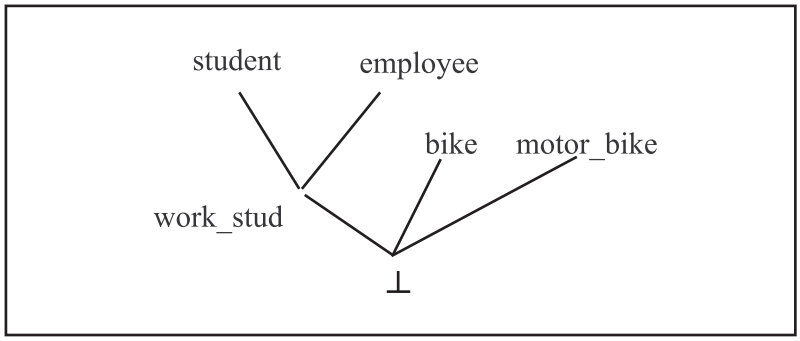

Figure 6: Signature for inheritance for the Example 4.1

Notice that in the above set we have the structural description of the entities only, without any hierarchical relation, although it exists between *work_stud* and *student*, *employee*. The hierarchical information among the entity names are modeled in a separate component, referred to as Signature for inheritance, that is illustrated in the next subsection.

4.2.3. The Signature for inheritance

In Life, the partially ordered set of sorts associated with a set of ψ-terms is referred to as a *Signature for inheritance* ($<$). A Signature for inheritance is axiomatically given and is always a lower semilattice (i.e, each pair of sorts has always a greatest common lower bound, possibly equal to $\bot$, i.e., the least element in the hierarchy) [18]. For instance, a complete representation of the Example 4.1 in Life requires to associate the above set of ψ-terms with the following Signature for inheritance:

$$\texttt{work\_stud} < \texttt{student}$$
$$\texttt{work\_stud} < \texttt{employee}$$
$$\bot < \texttt{work\_stud}$$
$$\bot < \texttt{bike}$$
$$\bot < \texttt{motor\_bike}$$

For the sake of simplicity, in the following we will omit the trivial relations involving the least element $\bot$. As already mentioned, in OOP inheritance hierarchies involve entities only, whereas in the Signature for inheritance entities and individuals may coexist. For instance, if $s1$ and $s2$ are two individuals, in the Signature for inheritance it is possible to specify that these individuals are students, as follows:

$$\texttt{s1} < \texttt{student}$$
$$\texttt{s2} < \texttt{student}$$

In the following two paragraphs we will illustrate how the notion of a Signature for inheritance is at the basis of the subtyping notion and the hierarchy processing defined in Life.

4.2.4. Subtyping

In Life, the notion of *subtyping* between ψ-terms is more restrictive than the one introduced in OOP (i.e., the structural subtyping). In particular, in Life we have a structural subtyping that can be established only among ψ-terms whose root sorts are related in the Signature for inheritance. For instance, consider the entity set:

`person`($name$ => `string`; age => `integer`)
`dog`($name$ => `string`; age => `integer`; $breed$ => `string`)

dog is not a subtype of *person*, unless this is explicitly declared in the Signature, that is:

`dog` < `person`.

Also in the case of common attribute labels, the associated ψ-terms are required to be hierarchically related. For instance, consider:

`person`($name$ => `string`; $friend$ => `person`)
`student`($name$ => `string`; $friend$ => `doglover`; $college$ => `integer`)

Then, the ψ-term *student* is not a subtype of *person* unless the related Signature for inheritance contains not only the relation between the root sorts:

`student` < `person`

but also the relation between the sorts:

`doglover` < `person`.

4.2.5. Hierarchy processing

In Life, the Signature for inheritance is mainly introduced in order to perform *unification*. Unification is a well-known operation in LP, and it is at the basis of *resolution*, one of the fundamental mechanisms of theorem proving [46, 31]. Unification is the process of determining whether two or more expressions can be made identical by appropriate substitutions for their variables. The unification algorithm proposed in Life is more general with respect to the standard unification algorithm proposed in the literature. In particular, in Life unification can be applied to ψ-terms with different root sorts and arity (number of properties), and the arguments of the ψ-terms are identified by their attribute labels. In Life, given any pair of ψ-terms, unification aims at deriving, when possible, a third ψ-term that is the greatest common lower bound according to subtyping. In this sense we can say that unification in Life is very similar to the normalization process in OOP. In particular, in the case of untagged ψ-terms, the unification algorithm proposed in [6] becomes quite simple and differs, essentially, from the normalization process in the following points:

- only the ψ-terms whose root sorts (principal types) have a non-trivial (i.e., different from $\perp$) common lower bound according to the Signature for inheritance can be unified;

- in the process, inheritance conflicts are solved according to the precedence relation given in the Signature for inheritance, i.e., if a non-trivial common lower bound of the conflicting root sorts exists. In particular, in both these cases, the greatest common lower bound is chosen.

For instance, in the Example 4.1, the unification between the ψ-terms whose principal types are *student* and *employee* can be applied, having them *work_stud* as greatest common lower bound (according to the related Signature for inheritance). However, this unification does not succeed due to the inheritance conflict generated by the attribute label *vehicle*. Such a conflict cannot be solved since the greatest common lower bound between the sorts *bike* and *motor_bike* is $\bot$.
Vice versa, suppose that the Signature for inheritance is replaced with the following one (where, as already mentioned, relations involving the least element $\bot$ has been omitted):

$$\texttt{work\_stud} < \texttt{student}$$
$$\texttt{work\_stud} < \texttt{employee}$$
$$\texttt{motor\_bike} < \texttt{bike}$$

In this case unification succeeds and, in particular, returns the ψ-term:

$$\texttt{work\_stud}(name => string; vehicle => motor\_bike;$$
$$college => string; salary => integer)$$

Notice that, in this example, the resulting ψ-term is a structural subtype (according to the OOP approach) of both the ψ-terms *student* and *employee*.

It is interesting to observe that unification succeeds also in the case of inheritance conflicts between sorts whose associated sets of attribute labels are not in structural subtyping. For instance, consider the Example 4.1, where *motor_bike* and *bike* are defined as follows:

$$\texttt{student}(name => string; vehicle => bike; college => string)$$
$$\texttt{employee}(name => string; vehicle => motor\_bike; salary => integer)$$
$$\texttt{bike}(maker => string; color => string; speed => integer)$$
$$\texttt{motor\_bike}(maker => string; color => string; power => string)$$

together with the Signature for inheritance defined above. In this case the unification between *student* and *employee* provides the ψ-term:

$$\texttt{work\_stud}(name => string; college => string; salary => integer;$$
$$vehicle => motor\_bike(maker => string; color => string;$$
$$power => string, speed => integer))$$

due to the presence of the precedence relation:

$$\texttt{motor\_bike} < \texttt{bike}$$

in the Signature. Therefore, we have a sort of double *motor_bike* definitions, although one explicitly given and the other one nested in another ψ-term.

In the Subsection 5.3, a methodology integrating the structural approach of OOP together with the approach proposed in the LP area is recalled [16], developed within the Object-oriented databases research field presented below.

5. Object-oriented databases

In this section, we will illustrate how the *Database* (DB) area has contributed to (static) conceptual modeling. This contribution can be traced back to the '70s, with the advent of *Semantic data modeling* [50, 76] and *Entity-relationship* [32, 14]. Along this line, an important impulse came from the *Object-oriented* (OO) approach [54]. *Object-oriented databases* (ODB) [53, 57, 58] represent a technology developed within the OO movement, that experienced its momentum at the beginning of the '90s. In particular, a strong motivation for the introduction of ODB resides in the advent of Object-oriented programming and the inadequacy of the relational databases [85, 82], from a logical point of view, in interacting with OO programs. This inadequacy arises from the mismatch due to the rich structures that can be manipulated in an OO program and the simpler structures available in the relational model (everything stored in a flat table, with columns representing only atomic attributes). For this reason, the first generation of *Object-oriented database management systems* ($ODBMS$) has been conceived for object persistence, i.e., to allow for the survival of objects managed by an OO program (written in an OO programming language). Among the proposed commercial products, we may mention: *Objectivity* [69], *Versant* [86], *Iris* [41], ODE [4], *Gemstone* [26], *Orion* [12], *Object Store* [70], *Visio* [47], $O2$ [11, 68]. In addition, there has been a number of academic prototypes, with a varying number of database features, such as *Galileo* (one of the first on the stage) [3], *Impress/TM* [9, 10], *Tycoon* [59], and *Logidata* [8]. Since the early '90s, when the first generation systems appeared, ODB technology has evolved and new systems have been developed, mainly driven by the success of *Java* [52]. In particular we may cite *Poet* [72], and *Jade* [51], on the commercial side, *PJama* [71], on the academic side. This (largely incomplete) list of systems gives the idea of the high fragmentation of the sector. This may be a reason for a market penetration slower than initially expected. Another reason can be seen in the fact that relational systems did not remain still, and started to evolve towards open and extensible solutions. The most remarkable one is represented by the *Object-relational* paradigm [80], that somehow integrates the two approaches, relational and OO, creating an hybrid paradigm. *Object-relational database management systems* ($ORDBMS$) appear able to preserve the investment on the relational side, while being suited for more advanced applications that require the modeling power of the object paradigm. Within such a complex scenario, it is very difficult to present a unique, comprehensive framework for the conceptual modeling of entities and hierarchies in object databases. We decided to recall two solutions of different nature. The first comes from the *Object database management group* ($ODMG$) [30], an international board for object database standards, the only one in the field tightly connected to the OMG standard organization (a larger organism that produced, among others, the $CORBA$ standard). The second is $O2$, a commercial system produced by O2 Technology. It was selected since it was among the first to be present in the ODB market segment and, at the same time, it is characterized by a sound theoretical basis.

5.1. The ODMG standard

The ODMG standard has been conceived to supply the ODB vendors with a reference model for an ODB system. Such a database model has a number of qualifying features aimed at helping the programmer in using an object database, from within a procedure developed using an OO programming language (in particular, the standard addresses *Smalltalk*, *C++*, and *Java*). Therefore, a large part of the standard is concerned with issues related to the programming level. Our focus is primarily on the structural modeling of entities, and their hierarchical organization. Therefore we will illustrate only a subset of the ODMG standard. In particular, we will address the part of the standard concerning the structural features of the proposed database model.

5.1.1. Entity

In ODMG, a *type* represents the fundamental notion of an entity. A type has an *extent*, represented by a set of instances. An instance can be an object or a literal. The extent of a literal type (such as *integer* or *string*) is axiomatically given and immutable. The extent of an object type (like *person* or *car*) is represented by data stored in a database, at a given time. An instance of an object type has a (mutable) state and an identifier (referred to as *oid*, see Section 2). Literals have no identifiers, but a complex literal (structure) may have a key. According to the ODMG standard, a type has an external *specification* and one or more *implementations*. A specification describes the *properties* and the *operations*. A type specification can be a *class* or an *interface*. Intuitively we may say that an interface cannot be instantiated, and only classes may have extents. The ODMG proposes a syntax, derived from OOP and the *Interface Definition Language* (defined by OMG). The syntax is intuitively presented through the examples below.

```
interface Person {
    ... properties ...
    ... operations ...
 };
class Employee {
    ... properties ...
    ... operations ...
 };
```

5.1.2. Entity expression and domain restriction

The modeling notion that corresponds to an entity expression, according to ODMG standard, is referred to as *type specification*. It is constructed, with regard to the static component, supplying the list of typed properties. Properties can be *attributes* or *relationships*. The former take values (essentially literals) when instantiated, the latter represent a bi-directional link with another type. (Note that when defining a relationship from *typeA* to *typeB*, the inverse, from *typeB* to *typeA* must be indicated as well.) A relationship may be one-to-one, one-to-many, or many-to-many.

```
class Employee {
    attribute string name;
    attribute short age;
```

```
    attribute enum gender {male,female};
    struct Address {string city, string street, short number};
    attribute set ⟨Phone_no⟩ phones;
    attribute Department dept;
};
interface Person {
    attribute string name;
    attribute Date birthDate;
};
```

Notice that an attribute can be implemented by a complex structure, like in the case of *Address* (further decomposed into sub-attributes), can be set-valued, like in the case of *phones* (other complex values, such as list or bag, are allowed), or may take an oid as its value, like in the case of *dept* (that will take as a value an oid from the elements of the extent of the *Department* type). However, in this case, where a property takes an oid as its value, referential integrity is not guaranteed. (Only relationships guarantee it.) A set of types with their definitions is referred to as *database schema*, the collection of the corresponding instances is a *database*. In the example, we also see the principal mechanisms for domain restriction. Attributes can be typed with basic types (such as *integer*, *string*, or *short*) or with complex types. In the latter case, we may have a structure (tuple), a multi-value, or an object reference. In the case of multi-valued attribute, however, there is no possibility of indicating cardinality constraints.

5.1.3. Refinement hierarchy

The ODMG object model allows types to be organized according to an inheritance-based, type-subtype hierarchy. In a schema we have two kinds of hierarchical relationships: ISA and EXTENDS. The former is required when the supertype is an interface, and allows inheritance of behaviour only. The subtype can be another interface or a class. The ISA hierarchy allows multiple inheritance to be modeled. However, name overriding is disallowed. Therefore, when two supertypes carry, say, the same attribute *phone*, the associated typing must agree. EXTENDS is used between classes and concerns both behaviour and state. This is a single inheritance relationship, but overriding is allowed. Syntactically, ISA is represented by a semicolon and EXTENDS is represented by the keyword *extends*.

```
interface Manager : Person {
    attribute set⟨Project⟩ manages;
};

class EmployeePerson extends Employee : Person {
    attribute Date hireDate;
    attribute Currency salary;
    relashionship Manager boss
        inverse Manager::subordinates;
};

class ManagerPerson extends EmployeePerson : Manager {
    attribute EmployeePerson secretary;
```

```
    relashionship set⟨Employee⟩ subordinates
        inverse Employee::boss;
};
```

Notice that in the above set of types (that is not a proper schema, since the *Department* type is undefined) only *EmployeePerson*, *ManagerPerson*, and *Employee* can be instantiated. Furthermore, there is a containment relationship between the extents of *Employee* and *EmployeePerson*, as well as between the extents of *EmployeePerson* and *ManagerPerson*.

5.1.4. Hierarchy processing

As anticipated, the refinement relation is established between two types: a more general supertype and a more specific subtype. Any subtype carries all the type information of its supertypes, up in the hierarchy. A subtype interface, as usual, may define characteristics in addition to those defined on its supertypes. Also the overriding mechanism applies here, therefore a characteristic defined on a supertype can be redefined, in more specialized terms, on the subtype. In the example above, the *EmployePerson* type inherits from the *Employee* class, by means of the EXTENDS relationship, and from the *Person* class, by means of the ISA relationship. The resulting structure for the *EmployeePerson* class is:

```
class EmployeePerson { // after inheritance
    attribute Date hireDate;
    attribute Currency salary;
    relashionship Manager boss
        inverse Manager::subordinates;
    attribute string name;
    attribute short age;
    attribute enum gender {male,female};
    struct Address {string city, string street, short number};
    attribute set ⟨Phone_no⟩ phones;
    attribute Department dept;
    attribute Date birthDate;
};
```

Notice that attributes and relationships can be intermixed at will.

5.2. The O2 approach

As already mentioned, the ODBMS O2 [11, 68] has been selected since it represents a significant industrial experience and, at the same time, is based on a sound methodological basis. The O2 is a complex software environment. Besides the data management solutions, it supports other functions, including programming environment, Object-oriented languages, user interface generation, web technology. In this section we focus on the object model adopted to represent a database schema, that includes different notions, such as, types, classes, relationships, and hierarchies.

5.2.1. Entity

In O2, both the notions of a type and a class are present; they denote, at extensional level, values and objects, respectively. Types can be atomic, such as *string* or *integer*, or complex. The latter are composed by using constructors such as: set, tuple, and list. Non-atomic values are complex literals, whose structure is defined by the corresponding types, and have no identity. (Although, at implementation level, structured values are given an identifier and are stored and retrieved as objects, mainly for homogeneity reasons, [11], p.45.) Untyped values are not allowed. Every class has a type associated that describes the structure of its instances. Objects, that are class instances, have a structure and an identity. Furthermore, objects are encapsulated, i.e., their internal structure is not visible and can be manipulated through methods only. Complex values have a visible structure and can be directly manipulated, e.g., with algebraic operators, like in the case of relational values. Given a finite set of class names **P** and an infinite set of constants D (literals and oids), the set of type expressions used in class definitions is expressed by the following abstract syntax, where τ is a type expression, P any subset of **P**, and $k \geq 0$:

$$\tau = \emptyset \mid P \mid D \mid [A : \tau_1, \ldots, A : \tau_k] \mid \{\tau\} \mid (\tau \vee \tau) \mid (\tau \wedge \tau)$$

5.2.2. Entity expressions and domain restriction

Here we illustrate the syntax of the type language of O2, focusing on the tuple structure. The first of the two following examples is a type, expressed in the data definition language of O2, used to define a *Person* class:

```
type tuple (name: string,
    address: (city: City, street: string),
    age: integer,
    hobbies: list("tennis", "stamps", "kites")).
```

As we see from the example, an attribute may take an atomic literal as value (as for *name*) or a complex literal, i.e., a structure (as for *address*). When an attribute is typed with a user defined type (*address.city*), a corresponding instance will take an oid as value. Multi-valued attributes are represented indicating if it consists of a set, bag, or list (the latter is the case of *hobbies*, in the example.) A class is defined as an association between a name and a type. Therefore, we may have a class *Person* corresponding to the above type.

```
add class Person
 type tuple (name: string,
    address: (city: City, street: string),
    age: integer,
    hobbies: list("tennis", "stamps", "kites"))
 with extension.
```

The last line indicates that the class *Person* can be directly populated by instances (i.e., objects). If this clause is not expressed, the class is assumed to be *abstract*, therefore it cannot be directly populated. (Similarly to interfaces in the ODMG standard, only its subclasses will be populated.)

A set of possible instances of the *Person* class are:

(#o1, [name: "Smith", address: [city: #o23, street: "main"], age:29,
hobbies: "tennis"])

(#o5, [name: "John", address: [city: #o23, street: "least"], age:20,
hobbies: "kites"])

The above instances clearly show that an object is a pair formed by an oid and a structure. To summarize, from a structural point of view, classes have a name and a type. Correspondingly, at instance level, objects have identifiers and values. Types are used to define the structure (and the behaviour, not addressed in this paper) of the objects.

5.2.3. Refinement hierarchy

Classes are created explicitly during database design and are organized according to an inheritance hierarchy. Complex types, created within class definitions, are unnamed and do not have their own inheritance hierarchy. However, type inheritance is a mechanism that takes place within a class inheritance hierarchy. Class inheritance, as usual, is mainly used to define an application (a database schema, in our case) incrementally. Imagine that we need to define the *Student* class where, besides what has been already specified for persons, we add two attributes: *college* and *pet*.

add class Student **inherits** Person
type tuple (college: string,
pet: set(PetAnimal))
with extension.

Therefore, a programmer uses *inherits* to define subclasses, giving only the extra attributes. If required, attributes already declared in the superclass may be redefined (this mechanism is referred to as *overriding*). In defining a subclass, the property of instance *substitutability* must be preserved. This means that any specialized object (i.e., instance of a subclass) can be seen as a more general object (i.e., legal instance of a superclass). For example, a student is a person in all respect. The property of substitutability is guaranteed by the subtyping relationship that must hold between the types associated with the class and subclass. Imagine that we need a graduate and student (*Gr_Student*) class, where we add a new attribute (the subject of the thesis) and we override the *pet* attribute:

add class Gr_Student **inherits** Student
type tuple (thesis_subject: string,
pet: set(Dog))
with extension.

Beyond the intuition, that says that a dog is indeed a pet animal, we are guaranteed that the above definition is *correct* only if the type associated with the *Dog* class is a subtype of that associated with the *PetAnimal* [42, 43]. On the line of the OOP approach, from a semantic point of view: "a type is a subtype of another if and only if every instance of the former is also

an instance of the latter" ([11], page 26). This property is often referred to as "set inclusion" semantics. From a syntactic point of view, the syntactic subtyping rule holds: "A tuple type is a subtype of another if it is more defined, that is, if it contains every attribute of its supertypes plus some new ones and/or it refines the type of some attributes of its supertype" ([11], page 27). Class hierarchy is tightly connected to type hierarchy, but the former is given by the programmer, while the latter is inferred by the system.

5.2.4. Hierarchy processing

We have seen that in O2, there are two different (but tightly related) sorts of hierarchies: type hierarchy and class hierarchy. There are two different (but tightly related) sorts of processing connected to the above mentioned hierarchies. Let us consider a class hierarchy: it is processed according to a rewrite mechanism that (recursively) applies inheritance. This mechanism follows the hierarchy and provides the subclasses with all the attributes of the superclasses, but avoids copying any attribute locally defined.

add class Gr_Student /* after inheritance
 type tuple (name: string,
 address: (city: City, street: string),
 age: integer,
 hobbies: list("tennis", "stamps", "kites"),
 college: string,
 thesis_subject: string,
 pet: set(Dog))
 with extension.

O2 supports multiple inheritance. However, it is allowed with a few restrictions, to avoid critical inheritance conflicts. When an attribute (or method) name is defined in two or more superclasses, the user either has to explicitly redefine the attribute or give the inheritance path ([11], p.28). Let us consider now a type hierarchy: it is processed in order to verify its well-formedness. To this end the subtyping relation is used. The verification takes place by using the subtyping syntactic rule defined above. In essence, it is required that two hierarchically related classes have "structurally compatible" types. If c and c' are two class names and σ is a mapping from class names to the related types, we require that:

$$c \prec c' \Rightarrow \sigma(c) \leq \sigma(c')$$

A class hierarchy is well-formed if for all classes c and c' the above rule is satisfied. A database schema is essentially a well-formed class hierarchy.

Both ODMG and O2 (as well as the great majority of strongly typed ODBMS) have adopted a structural approach to subtyping, which is inspired by the type theory illustrated in Subsection 4.1. There are cases in which this approach may accept a correct schema that may appear counter-intuitive. In the following subsection we recall a proposal that integrates the structural and naming approaches, where a notion of correct schema (*well-formed* schema) that is closer to a more intuitive view of the reality is proposed.

5.3. Integrating naming and structural approaches in ODB

In this subsection, we start from a comparative analysis of the inheritance processing approaches adopted in the OOP and LP fields (illustrated in Section 4), in order to recall an integrated solution for inheritance hierarchy processing presented in [16]. We have seen that each of the mentioned approaches presents some interesting aspects but also some drawbacks. By summarizing, we have seen that with the structural approach, different entities that exhibit the same structure are considered equivalent (i.e., entity and property names do not carry any semantics). On the other hand, with the naming approach, hierarchies are considered independently of the definitions of the conflicting types, therefore generating double entity definitions. For this reason, in [16] a new methodology is presented, aiming at solving the above mentioned problems by providing a balanced solution that takes into account both the entity structures and the name precedence relation (as the one proposed in Life with the Signature for inheritance). According to such a methodology, inheritance is performed by using:

- the inheritance mechanism of the structural approach, and
- the inheritance conflicts resolution that makes use of a declared hierarchy of entity tags. Such a hierarchy is not arbitrarily given as, for instance, in Life, but it is derived from the refinement hierarchy declared in the entity definitions (by using, for instance, the **and** or **ISA** construct).

Therefore, an entity set organized according to a refinement hierarchy is always coupled with a hierarchy of entity tags. The ordering relation imposed on entity tags is referred to as $DescOf$ ($DescendantOf$). The pair (E,$DescOf$) formed by the set E of entity tags of the schema and the $DescOf$ relation is a partially ordered set. In the proposed approach, inheritance conflicts are solved making use of the $DescOf$ relation derived from the schema, when possible. In the cases where conflicts cannot be solved, the method provides suggestions about possible modifications of the schema and the $DescOf$ relation aimed at removing the inheritance conflicts. We will see that such suggestions aim at guaranteeing the formal property of *consistent refinement hierarchy* recalled in Subsection 2.5 (here referred to as *schema well-formedness*), that the entity set, once transformed into expanded form, must verify. Below, the main points of this methodology are briefly recalled.

5.3.1. Interface

According to the ODMG standard, in [16] a schema contains definitions for *literal* types (i.e., types of complex values) and *interfaces* (entities). Literal types are constructed from atomic types such as *integer, string* by using constructors, such as *tuple* or *set*. Their domains are fixed, essentially predefined. Interfaces are defined (i.e., user-defined), and their domains are populated by explicit object creation requests (i.e., by populating a database).

5.3.2. Interface definition

An interface definition contains a *name* and a set of *typed properties* (*tuple*). The types for properties are: (i) atomic types (such as *string* or *integer*), (ii) interface names (such as *person* or *car*), (iii) tuples (i.e., nested tuples). In case (i) and (ii) properties are referred to as *attributes* and *relationships*, respectively. An interface definition can also have an *inheritance component*, containing the names of the superinterfaces (superentities) from which typed properties are

inherited. The inheritance hierarchy is specified by the **and** constructor. However, in order to reduce the number of different formalisms, here we will use the neutral syntax of TQL, i.e., the **ISA** constructor. The tuple of an interface definition is referred to as the *explicit component* of the definition. A database *schema* is a set of interface definitions, where each interface name is uniquely defined, every definition contains only defined interface names, and inheritance is acyclic. A schema is in *explicit form* iff all its definitions contain explicit components only, otherwise, if at least one of them contains an inheritance component, the schema is said to be in *implicit form*. The tuple of an interface definition in explicit form, whose name is τ, is indicated as $i(\tau)$.

In [16], following the approach adopted in Life, it was proposed to keep the distinction among structurally similar but semantically different types by means of a "branding" mechanism. Branding enriches the tuple constructor. With respect to plain tuples that can be compared (and manipulated) by considering only the properties (e.g., by making the union of typed properties in performing inheritance), branded tuples require "compatibility" (that holds along a refinement hierarchy path). Therefore, it is not possible to perform the union of typed properties of tuples with incompatible branding. A branded tuple constructor has the form:

$$\tau := [_{\tau}\ p_i : \zeta_i]$$

where each "$p_i : \zeta_i$", $1 \leq i \leq n$, is a typed property. For instance, the interfaces:

$$\texttt{person} := [_{person}\ name : \texttt{string}, age : \texttt{integer}]$$
$$\texttt{dog} := [_{dog}\ name : \texttt{string}, age : \texttt{integer}]$$

can be distinguished as different interfaces, even if they are structurally equivalent (i.e., their sets of typed properties coincide).

However, notice that in this approach the designer is not required to explicitly specify branding in tuples, since it is automatically derived from the interface definition (in fact, the tuple branding is represented by a tag that coincides with the interface name).

On the line of the Life approach, we will see in the next paragraphs that the branding mechanism has a direct impact on subtyping. In this proposal the *specialized type* relation has been introduced to take into account branding. Before presenting such a relation, we need to define the $DescOf$ relation.

5.3.3. Refinement hierarchy

An implicit schema, i.e., a schema containing **ISA** constructs, induces a precedence relation on interface names, referred to as *directDesc* (*directDesc-endant*). The *directDesc* relation is defined among the names of the interfaces being defined and each of its superinterfaces, as follows.
If τ and τ_i are interface names:

$directDesc(\tau, \tau_i)$, if τ_i is used in the
inheritance component of the definition of τ.

Starting from the *directDesc*, the $DescOf$ ($DescendantOf$) relation is defined as the reflexive and transitive closure of the *directDesc* relation.

For instance, consider the interfaces of the Example 4.1, specified by using the TQL language as follows:

$\texttt{student} := [name : \texttt{string}, vehicle : \texttt{bike}, college : \texttt{string}]$
$\texttt{employee} := [name : \texttt{string}, vehicle : \texttt{motor\_bike}, salary : \texttt{integer}]$
$\texttt{bike} := [maker : \texttt{string}, color : \texttt{string}]$
$\texttt{motor\_bike} = [maker : \texttt{string}, color : \texttt{string}, power : \texttt{string}]$
$\texttt{work\_stud} := \textbf{ISA}\ \texttt{student employee}$

then, the *directDesc* induced from this schema is:

$directDesc(work\_stud, student)$
$directDesc(work\_stud, employee)$

In the following, given a schema in explicit form, say Σ, we define a *full explicit* schema, say $\mathcal{S}$, as a pair $\mathcal{S} = (\Sigma, descOf_{\mathcal{S}})$, where $descOf_{\mathcal{S}}$ is a partial order on the interface names of Σ, called the *inheritance relation* of the schema $\mathcal{S}$.

This notion will be used below, and will allow us to illustrate the specialized type relation and hierarchy processing of this proposal.

5.3.4. Specialized type relation

For a full explicit schema $\mathcal{S} = (\Sigma, descOf_{\mathcal{S}})$ as defined above, the *specialized type* relation ($<:$) is defined on types over Σ as follows:

- Any type: $\tau <: \tau$.
- Interfaces: $\sigma_1 <: \sigma_2$ if $descOf_{\mathcal{S}}(\sigma_1, \sigma_2)$
- Tuples: $[_\tau\ p_i : \zeta_i]_{1 \le i \le n+m} <: [_\sigma\ p_i : \eta_i]_{1 \le i \le n}$
  if $descOf_{\mathcal{S}}(\tau, \sigma)$ and $\zeta_i <: \eta_i$ for $1 \le i \le n$.
- Sets: $\{\tau_1\} <: \{\tau_2\}$ if $\tau_1 <: \tau_2$.

The above *Tuples* case corresponds to a "branded structural subtyping", that is, subtyping as defined in the structural approach enriched with the branding mechanism. Conversely, the *Interfaces* case makes the specialized type relation very similar to the naming approach proposal (see Subsection 4.2).

For instance, with regard to the type interfaces *dog* and *person* of Subsection 5.3.2:
$dog <: person$
if:
$descOf(dog, person)$
holds. The same holds in the case of the example given in Subsection 4.1, that is recalled below:

$\texttt{person} := [name : \texttt{string}, age : \texttt{integer}]$
$\texttt{dog} := [name : \texttt{string}, age : \texttt{integer}, breed : \texttt{string}]$.

5.3.5. Hierarchy processing

In the absence of inheritance conflicts, we have seen that, essentially, the inheritance processes as proposed in the structural and naming approaches coincide. Conversely, inheritance conflicts resolution yields to different results. In this proposal an inheritance mechanism has been defined that includes an algorithm based on branding and the $DescOf$ relation (referred to as the *Expand* algorithm). It is briefly summarized below.

Given an implicit schema Σ, consider the set of interface names of Σ, say N, and the $DescOf$ relation induced by Σ. Then, for each inheritance conflict, the partially ordered set $P = (N, DescOf)$ is analyzed. If the conflicting interface names admit an interface name in P that is their greatest common lower bound, then the *Expand* algorithm succeeds and such a name is inherited in the expanded definition. Then, the expanded schema, together with the $DescOf$ relation, is said to be *well-formed*, i.e., for all interface names τ_1, τ_2:

if $DescOf(\tau_1, \tau_2)$ then $i(\tau_1) \leq: i(\tau_2)$.

Otherwise, the algorithm provides suggestions about possible modifications of the $DescOf$ (i.e., the existing inheritance hierarchy), and the related interface definitions. These suggestions may concern the introduction of interface definitions in the schema and interface names in the $DescOf$ relation.

Let us show a few examples.

In the case of inconsistent refinement hierarchy, as for instance the one illustrated in Section 2.5, the only suggestion that is given by the *Expand* algorithm is to reformulate the interface definitions. In fact, this kind of conflict, that takes place between *string* and *integer* (that are types with disjoint domains), is referred to as *unamendable*. Conversely, in the case of *amendable* conflicts, the algorithm suggests one or more possible solutions, allowing the designer maximum flexibility in choosing one that suits his/her needs. For instance, consider the schema:

`student` := [*name* : `string`, *vehicle* : `push_bike`, *college* : `string`]
`employee` := [*name* : `string`, *vehicle* : `motor_bike`, *salary* : `integer`]
`motor_bike` := [*maker* : `string`, *power* : `string`]
`push_bike` := [*maker* : `string`, *speed* : `string`]
`work_stud` := **ISA** `student employee`

whose $DirectDesc$ relation is the following:

$DirectDesc(work\_stud, student)$
$DirectDesc(work\_stud, employee)$.

In this case, the inheritance conflict due to the *vehicle* property cannot be solved since *motor_bike* and *push_bike* do not admit any (greatest) common lower bound, according to the $DescOf$ relation. However, this conflict is *amendable*, in the sense that it could be solved by simply introducing a new interface in the schema whose name is, for instance, *moped*, that inherits from both the conflicting interfaces *motor_bike* and *push_bike*, i.e.:

`moped` := **ISA** `motor_bike push_bike`

This modification allows the inheritance process to succeed, and the following explicit schema is obtained:

`student` := [*name* : `string`, *vehicle* : `push_bike`, *college* : `string`]
`employee` := [*name* : `string`, *vehicle* : `motor_bike`, *salary* : `integer`]
`motor_bike` := [*maker* : `string`, *power* : `string`]
`push_bike` := [*maker* : `string`, *speed* : `string`]
`work_stud` := [*name* : `string`, *vehicle* : `moped`,
college : `string`, *salary* : `integer`]
`moped` := [*maker* : `string`, *power* : `string`, *speed* : `string`]

The *directDesc* relation is also modified as represented in Figure 7 (it is an extension of the *directDesc* relation derived from the original schema).

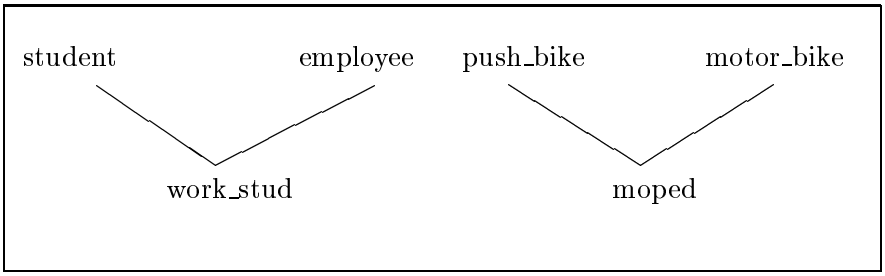

Figure 7

Notice that the full explicit schema obtained from the resulting explicit schema and the modified $DescOf$ relation is a well-formed schema.

In the case of the Example 4.1 (see Section 4.1), the conflict between *bike* and *motor_bike* can be solved by extending the *DirectDesc* relation with the pair:

$$DirectDesc(motor\_bike, bike)$$

that is, by modifying the *DirectDesc* relation as shown in Figure 8.

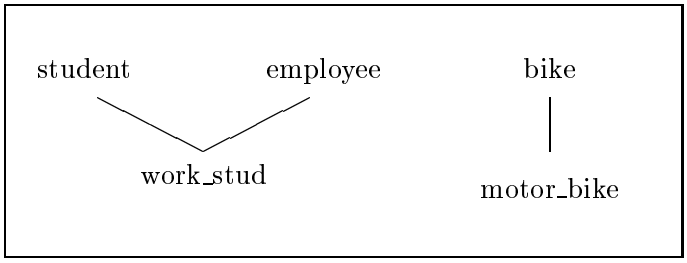

Figure 8

Therefore the conflict generated by the property *vehicle* can be solved by inheriting the *motor_bike* name interface as follows:

$\texttt{work\_stud} := [name : \texttt{string}, vehicle : \texttt{motor\_bike}, college : \texttt{string},$
$salary : \texttt{integer}]$

Notice that in all the above examples, the obtained full explicit schema, consisting of the expanded schema and the $DescOf$ relation, is well-formed. Indeed, it is possible to prove that:

if the Expand algorithm succeeds, than the resulting full explicit schema is well-formed.

In [16], particular attention is paid to the case in which the partially ordered set does not provide unique solutions. Such ambiguous situations arise, for instance, in the presence of conflicting interface names that, instead of having a greatest lower bound, admit two or more maximal lower bounds (i.e., the partially ordered set is not a lower semilattice). Also in these cases, the ambiguity can be removed by suitably modifying the schema and the $DescOf$ relation (for a deeper understanding of this aspect of the proposal, see details in the mentioned paper).

It is the opinion of the authors that the approach just recalled represents a well balanced solution for conceptual modeling of refinement hierarchies. This is obtained with a balanced blend of the formal treatment of entity structures and the axiomatic, given, partially ordered set of domain dependent entity names. This solution appears to be more intuitive in a larger number of cases than the previous ones.

6. Conclusion

In this paper we surveyed the basic notions of structural conceptual modeling, with particular attention to refinement hierarchies, and how such notions have been addressed in three important areas of Computer science: Artificial intelligence, Programming languages, and Databases. Refinement hierarchies and, more generally, the modeling of the static aspects of a problem domain, have attracted a lot of attention and many relevant results can be found in the literature. The intent was not to be exhaustive, but rather to select a few representative proposals in each area. The presentation of a unifying analysis framework was not easy. In fact, due to the different backgrounds, starting points, and evolution lines, the different areas are treating the problem in different ways, with different approaches and terminology. Therefore, the first effort in this survey has been that of producing a unifying frame to allow a consistent view of the different proposals analyzed. The unifying frame has been constructed identifying the primary modeling notions. Firstly, the central notion of an entity has been addressed, reporting about the constructs and the elements necessary to define it. An entity definition consists in the information structure, constructed supplying its properties and relationships with other entities. But the key issue of this survey concerned the organization of entities according to a refinement hierarchy, how a hierarchy is defined and what kind of processing is performed on it. The paper has been structured according to the three above mentioned areas, that gather the analyzed proposals. Here, in the concluding section, we aim at summarizing the gist of the analysis presented in the different sections, building a synoptic table. We believe that such a table will help the reader to better understand the different proposals through a comparative view. The synoptic table is organized as follows. The columns are labeled with the references to the analyzed proposals and the rows refer to the modeling notions. In particular, on the columns we have:

- **DL**, for Description logics;
- **CL**, for Concept lattice;
- **OOP**, for Object-oriented programming, as seen in the proposal of Cardelli;
- **LP**, for Logic programming, as seen in the proposal of Ait-Kaci;
- **ODMG**, for the object database standard;
- **O2**, for the object database model of the homonymous management system;
- **Int-App**, for an integrated approach, concerning the proposal that integrates structural and naming approaches.

The rows are labeled with the modeling notions that have been used to analyze the different proposals. In particular, on the rows we have:

- **Entity**, as a primary notion. Then in each cell the term that in the corresponding proposal denotes this notion is given.
- **Property**, represents the basic notion used to define an entity. Some proposals require a *domain restriction* to be specified (e.g., by typing), other accept domain restriction as an option, yet other do not consider this information at all. Terminology is quite stable and the main accepted difference is between *attribute*, to define a property that is instantiated with a literal, and *relationship*, when the property represents a reference to another entity. The main singularity is in DL where *role* is used.
- **Entity expression**, is the term used to indicate the definition of an entity. Here there are a number of differences that should be pointed out. In terms of syntax there are noteworthy differences. The most intuitive (but also elementary) is represented by the set of attributes used in Concept lattices. The most complex appears to be the ψ-*term* of LP, where individual instances an even *coreference constraints* can be expressed.
- **Hierarchy definition**. Also here, besides the syntactic differences, we have the option represented by attributes and objects set inclusions, as proposed by CL, and the one proposed in PL, that introduces the notion of *Signature for inheritance*. The latter is the only proposal that considers the entity names as carrying a semantic import (although, axiomatically given).
- **Hierarchy processing**. The main operation here is represented by inheritance, that is proposed by all but DL. The latter is not really interested in the information structure of the concepts, but rather in the operation of organizing concepts along the hierarchy (i.e., *Taxonomic reasoning*). Many solutions consider also the possibility of verifying the correctness of a given hierarchy.

In conclusion, a general impression is that the rich diversity of syntax, terminology, and basic assumptions that we presented in this survey paper will remain, and there is not a sign that, in structural conceptual modeling, a unifying theory will emerge soon. Nevertheless, there are significant signs that the need for conceptual modeling, and mechanisms for managing the increasing complexity of domains to be analyzed and systems to be built, is expanding. *Abstraction*

(and, therefore, its dual mechanism: *refinement*) is one of the primary mechanisms to tackle complexity (another is *partitioning*, that leads to decomposition hierarchies, not addressed in this paper). Therefore, we believe that new formalisms, methods and tools will be proposed in the future. In any case, they may benefit from the wealth of proposals coming from previous experiences, that represent a very rich set of possible solutions that new proposals should carefully analyze.

	DL	**CL**	**OOP**	**LP**
Entity	- concept	- concept	- record type	- ψ-term
Property	- role (unrestr./ restricted)	- unrestricted attribute	- attribute/ relationship	- property
Entity expression	- concept expression	- sets of objects/ attributes	- type definition	- attributed ψ-term
Hierarchy definition	- AND construct	- inheritance relation	- AND construct	- Signature for inheritance
Hierarchy processing	- subsumption	- inheritance graph	- subtyping - normalization - type inference - type checking	- subtyping - unification

	ODMG	**O2**	**Int-App**
Entity	- type	- type/class	- interface
Property	- attribute/ relationship	- attribute	- property
Entity expression	- class/interface	- type expression	- interface definition
Hierarchy definition	- ISA/*extends* constructs	- *inherits* construct	- ISA construct
Hierarchy processing	- restricted inheritance	- restricted inheritance	- specialized type relation - expansion

Synoptic Table

References

[1] M.Abadi, L.Cardelli; *A Theory of Objects*; Springer, 1996.

[2] R.M.Amadio, L.Cardelli; *Subtyping recursive types*; ACM Transactions on Programming Languages and Systems 15(4); pp.575-631, 1993.

[3] A.Albano, L.Cardelli, R.Orsini; *Galileo: a strongly-typed, interactive conceptual languge*; ACM Trans. on Database Systems 10, pp.230-260, June 1985.

[4] R.Agrawal, N.H.Gehani; *ODE (Object Database and Environment): The Language and the Data Model*; Proc. ACM-SIGMOD 1989, Int. Conf. Management of Data, Portland, Oregon, pp.36-45, May-June 1989.

[5] S.Abiteboul, R.Hull, V.Vianu; *Foundations of Databases*; Addison Wesley, 1995.

[6] H.Ait-Kaci, A.Podelski; *Towards a Meaning of Life*; J. of Logic Programming, 16, pp.195-234, 1993.

[7] H.Ait-Kaci; *An Algebraic Semantics Approach to the Effective Resolution of Type Equations*; Theoretical Computer Science 45, pp. 293-351, Noth-Holland, 1986.

[8] P.Atzeni, F.Cacace, S.Ceri, L.Tanca; *The Logidata+ Model*; in "Logidata+: Deductive Databases with Complex Objects", P.Atzeni (Ed.), Lecture Notes in Computer Science (LNCS) 701, pp.20-29, Springer-Verlag, 1993.

[9] H.Balsters, R.A.de By, C.C.de Vreeze; *The TM Manual*; University of Twente, Technical Report INF92-81; Enschede, 1992.

[10] H.Balsters, M.M.Fokkinga; *Subtyping can have a simple semantics*; Theoretical Computer Science 87, pp.81-96, September 1991.

[11] F.Bancilhon, C.Delobel, P.Kanellakis (Ed.s); *Building an Object-Oriented Database System: The Story of O2*, Morgan Kaufman, 1992.

[12] J.Banerjee, H.Chou, J.F.Garza, W.Kim, D.Woelk, N.Ballou, H.J.Kim; *Data Model Issues for Object-Oriented Applications*; in "Readings in Object-Oriented Database Systems", S.B.Zdonik and D.Maier (Eds.), pp.197-208, Morgan Kaufmann; San Mateo, CA, 1990.

[13] C.Batini, S.Ceri, S.B.Navathe; *Conceptual Database Design - An entity-relationship approach*; Benjamin-Cummings, Redwood City CA, 1992.

[14] C.Batini, M.Lenzerini, S.B.Navathe; *A comparative analysis of methodologies for Database schema integration*; ACM Computing Surveys 18, 4, pp.323-364, 1986.

[15] C.Beeri; *A formal approach to object-oriented databases*; Data & Knowledge Engineering 5; pp. 353-382; North-Holland, 1990.

[16] C.Beeri, A. Formica, M. Missikoff; *Inheritance Hierarchy Design in Object-Oriented Databases*; Data & Knowledge Engineering (DKE), Vol.30, No.3, pp.191-216, July 1999.

[17] C.Beeri, T.Milo; *Subtyping in OODB's*; ACM PODS'91, May 29-31, Denver, Colorado, pp.300-314, 1991.

[18] G.Birkoff; *Lattice Theory*, Amer. Math. Soc. Providence, R.I., 1967.

[19] S.Bergamaschi, C.Sartori; *On Taxonomic Reasoning in Conceptual Design*. TODS 17(3), pp.385-422, 1992.

[20] G.Birkoff. *Lattices and their applications*, Bull. Amer. Math. Soc. 44, pp.793-800, 1938.

[21] R.J.Brachman; *What IS-A Is and Isn't: An Analysis of Taxonomic Links in Semantic Networks*; IEEE Computer, pp.30-36, October 1983.

[22] R.J.Brachman; *On the epistemological status of semantic networks*; in "Associative Networks - Representation and use of Knowledge by Computers", N.V.Findler (Ed.); Academic Press, New York, 1979.

[23] R.J.Brachman, H.J.Levesque; *The Tractability of Subsumption in Frame-Based Description Languages*; pp. 34-37, AAAI 1984.

[24] R.J.Brachman, H.J.Levesque, R.Reiter; *Introduction to the Special Volume on Knowledge Representation*; Artificial Intelligence 49(1-3), 1991.

[25] R.J.Brachman, J.Schmolze; *An overview of the KL-ONE knowledge representation system*; Cognitive Science 9, 2, pp.171-216, 1985.

[26] R.Bretl et al.; *The GemStone data management system*; in "Object-Oriented Concepts, Applications, and Databases", W.Kim and F.Lochovsky (Eds.), Reading, MA, Addison-Wesley, 1989.

[27] M.L.Brodie, J.Mylopoulos, J.W.Schmidt (Eds.); *On Conceptual Modelling: Perspectives from Artificial Intelligence, Databases, and Programming Languages*, Springer-Verlag, 1984.

[28] L.Cardelli; *A Semantics of Multiple Inheritance*; Info.&Comp., 76, pp. 138-164; 1988 (Preliminary version in LNCS 173, Springer-Verlag, 1984).

[29] B.Carpenter; *The Logic of Typed Feature Structures*; Cambridge University Press, October 1992.

[30] R.G.G.Cattel et Al.; *The Object Data Standard: ODMG 3.0*; Academic Press, 1999.

[31] C.Chang, R.C.Lee; *Symbolic Logic and Mechanical Theorem Proving*; Academic Press, 1987.

[32] P.P.Chen; *The entity-relationship model - Toward a unified view of data*; ACM Transactions on Database Systems, 1, 1, pp.9-36, 1976.

[33] W.F.Clocksin, C.S.Mellish; *Programming in Prolog*, 4th Edition; Springer Verlag, 1994.

[34] W.R.Cook, W.L.Hill, P.S.Canning; *Inheritance Is Not Subtyping*; Proc. of ACM Int. Conf. on Principles of Programming Languages (POPL'90); pp. 125-135, 1990.

[35] W.Cook, J.Palsberg; *A Denotational Semantics of Inheritance and its Correctness*; Proc. of Int. Conf. on Object-Oriented Programming Systems Languages and Applications (OOPSLA'89), pp. 433-443, October 1989.

[36] L.Cardelli, P.Wegner; *On Understanding Types, Data Abstraction, and Polymorphism*; Computing Surveys, Vol.17, No.4, December 1985.

[37] R.Godin, R.Missaoui, H.Alaoui; *Incremental Concept Formation algorithms based on Galois (concept) lattices*; in Computational Intelligence, Vol.11, No.2, pp. 246-265, 1995.

[38] F.M.Donini, M.Lenzerini, D.Nardi, W.Nutt; *The complexity of concept languages*; in Proc. of the second Int. Conf. on "Principles of Knowledge Representation and Reasoning" J.A.Allen, R.E.Fikes, E.Sandewall Eds., pp.151-162, 1991.

[39] F.M.Donini, M.Lenzerini, D.Nardi, A.Schaerf; *Reasoning in Description Logics*; A Great Collection, U.Gnowho and U.Gnowho-Else Eds., CSLI Publ., 1995.

[40] B.Eckel; *Thinking in C++*; 2nd Edition, Prentice Hall, 2000.

[41] D.H.Fishman et al.; *Iris: An Object-Oriented Database Management System*; ACM Trans. Office Inform. Syst., Vol.5, No.1, pp.48-69, January 1987.

[42] A.Formica, H.D.Groger, M.Missikoff; *Object-Oriented Database Schema Analysis and Inheritance Processing: a Graph-Theoretic Approach*; Data & Knowledge Engineering, Vol. 24, No. 2, pp. 157-181, North-Holland, 1997.

[43] A.Formica, H.D.Groger, M.Missikoff; *An Efficient Method For Checking Object-Oriented Database Schema Correctness*; ACM Transactions on Database Systems (TODS), Vol.23, No.3, pp. 333-369, September 1998.

[44] A.Formica, M.Missikoff; *Integrity Constraints Representation in Object-Oriented Databases*; in "Information and Knowledge Management", T.W.Finin, C.K.Nicholas, Y.Yesha (Eds.), Lecture Notes in Computer Science (LNCS) 752, pp. 69-85, Springer-Verlag, 1993.

[45] A. Formica, M. Missikoff; *Integrity Constraints and Structural Modeling in Object-Oriented Databases: a Unified Approach*; Int. Journal of Computers & Applications, Vol.21, No.3, pp. 89-99, 1999.

[46] M.R.Genesereth, N.J.Nilsson; *Logical Foundations of Artificial Intelligence*; Morgan Kaufmann; Los Altos, CA, 1987.

[47] R.Grabowski; *Learn Visio 5.0*; Wordware Publishing, 1998.

[48] T.A.Halpin, H.A.Proper; *Subtyping and polymorphism in object-role modelling*; Data & Knowledge Engineering 15, pp.251-281, 1995.

[49] J.Heinsohn, D.Kudenko, B.Nebel, H.Profitlich; *An empirical analysis of terminological representation systems*; Artificial Intelligence 68, pp. 367-397, 1994.

[50] R.Hull, R.King; *Semantic Database Modeling: Survey, Applications, and Research Issues*; ACM Computing Surveys, Vol.19, No.3, September 1987.

[51] http://www.discoverjade.com/

[52] http://www.java.sun.com

[53] S.Khoshafian; *Object-Oriented Databases*; John Wiley, 1993.

[54] S.Khoshafian, R.Abnous; *Object-Orientation - Concepts, Languages, Databases, User Interfaces*; Wiley, New York, 1990.

[55] S.N.Khoshafian, G.P.Copeland; *Object Identity*; Proc. of Int. Conf. on Object-Oriented Programming Systems Languages and Applications (OOPSLA'86), September 1986.

[56] M.Kifer, G.Lausen; *F-Logic: A Higher-Order Languge for Reasoning about Objects, Inheritance, and scheme*; Proc. ACM-SIGMOD 1989, In'l Conf. Management of Data, Portland, Oregon, pp.134-146, May 1989.

[57] W.Kim; *Introduction to Object-Oriented Databases*; Computer Systems, The MIT Press, Cambridge, 1990.

[58] W.Kim; *Object-Oriented Databases: Definition and Research Directions*; IEEE Transactions on Knowledge and Data Engineering, Vol.2, No.3, September 1990.

[59] F.Matthes, J.W.Schmidt; *System Construction in the Tycoon Environment: Architectures, Interfaces and Gateways*; in Proc. of Euro-Arch'93 Congress; Springer-Verlag, October 1993.

[60] E.Mendelson; *Introduction to Mathematical Logic*; D.Van Nostrand Comapny - Princeton, New Jersey; 1964.

[61] B.Meyer; *Eiffel: The Language*, Prentice Hall, 2000.

[62] M.Missikoff, M.Toiati; *MOSAICO - A System for Conceptual Modeling and Rapid Prototyping of Object-Oriented Database Applications*; Proc. of ACM SIGMOD Conference, Minneapolis, p.508, 24-27 May, 1994.

[63] M. Missikoff, M. Scholl; *An Algorithm for Insertion into a Lattice: Application to Type Classification*; in "Foundation of Data Organization and Algorithms", Lecture Notes in Computer Science n.367, Springer Verlag, Heidelberg, pp 64-82, 1989.

[64] C.Moss; *Prolog++: The power of Object-Oriented and Logic Programming*; Addison-Wesley; 1994.

[65] B.Nebel; *Computational complexity of terminological reasoning in BACK*; Artificial Intelligence 34, 3, pp.371-383, 1988.

[66] B.Nebel; *Terminological Cycles: Semantics and Computational Properties*; in "Principles of Semantic Networks", J.F.Sowa (Ed.); Morgan Kaufmann, 1991.

[67] B.Nebel; *Terminological reasoning is inherently intractable*; Artificial Intelligence 43 (2), pp.235-249, 1990.

[68] http://www.inria.fr/RII/O2-eng.html

[69] http://www.objectivity.com/

[70] http://www.odi.com/products/objectstore.html

[71] http://www.sun.com/research/forest/opj.main.html

[72] http://www.poet.com

[73] M.R.Quillian; *Semantic Memory*; in "Semantic Inforamtion Processing"; M.Minsky (Ed.), pp. 227-270, MIT Press, Cambridge, MA, 1968.

[74] K.Schild; *Tractable Reasoning in a Universal Description Logic*; KRDB, 1994.

[75] S.C.Shapiro (Ed.); *Encyclopedia of Artificial Intelligence*; John Wiley & Sons, 1987.

[76] J.M.Smith, D.C.P.Smith; *Database abstractions: Aggregation and generalization*; ACM Transactions on Database Systems 2, 2, pp.105-133, 1977.

[77] M.Schmidt-Schauβ; *Subsumption in KL-ONE is undecidable*; in Proc. of the first Int. Conf. on "Principles of Knowledge Representation and Reasoning" R.J.Brachman, H.J.Levesque, R.Reiter Eds., pp.421-431, 1989.

[78] J.F.Sowa; *Knowledge Representation - Logica, Philosophical, and Computational Foundations*; Brooks/Cole, Thomson Learning, 2000.

[79] J.F.Sowa; *Conceptual Graph Standard and Extension*; ICCS 3-14, 1998.

[80] M.Stonebraker, P.Brown; *Object-Relational DBMSs*; Morgan Kaufmann Publisher, 1999.

[81] A.Taivalsaari; *On the Notion of Inheritance*; ACM Compting Surveys 28, 3, pp.438-479, 1996.

[82] T.J.Teorey, D.Yang, J.P.Fry; *A Logical design methodology for relational databases using the extended entity-relationship model*; ACM Compting Surveys 18, 2, pp.197-222, 1986.

[83] D.C. Tsichritzis, F.H. Lochovsky; *Data Models*; Prentice-Hall, Englewood Cliffs, N.J., 1982.

[84] M.Tsukamoto, S.Nishio, M.Fujio; *DOT: A Term Representation using DOT Algebra for Knowledge-bases*; Deductive and Object-Oriented Databases, Lecture Notes in Computer Science (LNCS) 566, Springer-Verlag, 1991.

[85] J.D.Ullman; *Principles of Database and Knowledge-Base Systems*; vol.I; Computer Science Press; 1988.

[86] *Versant Language Interfaces Manual*; Version 1.6, Versant Object Technology, Menlo Park, CA, 1991.

[87] P.Wegner, S.B.Zdonik; *Inheritance as an Incremental Modification Mechanism or what Like is and isn't Like*; Proc. of European Conference on Object-Oriented Programming (ECOOP'88, S.Gjessing and K.Nygaard (Eds.), Springer-Verlag, LNCS 322, 1988.

[88] P.Wegner, S.B.Zdonik; *Models of Inheritance*; Workshop on Database Programming Languages (DBPL), Oregon, Morgan Kaufmann, pp. 248-255, 1989.

[89] R.Wille; *Restructuring lattice theory: an approach based on hierarchies of concepts*; Sym. on Ordered Sets, I.Rival Ed., Reidel, Dordrecht, Boston, 1982.

[90] P.H.Winston, *On to Smalltalk*, Addison-Wesley, 1997.

[91] W.A.Woods, J.G.Schmolze; *The KL-ONE family*; in F.W.Lehmann Ed., "Semantic Networks in Artificial Intelligence", pp.133-178, Pergamon Press, 1992.

[92] A.Yahia, L.Lakhal, R.Cicchetti, J.P.Bordat; *iO2 An Algorithmic Method for Building Inheritance Graphs in Object Database Design*; Proc. of ER'96, Cottbus, Germany, October 1996.

[93] K.Yokota, H.Tsuda, Y.Morita; *Specific Features of a Deductive Object-Oriented Database Language Quixote*; Proc. of the Workshop on Combining Declarative and Object-Oriented Databases (DOOD Workshop'93), pp 89-99, Washington, USA, 1993.

[94] R.Zicari; *A Framework for Schema Updates in an Object-Oriented Database System*; in "Building an Object-Oriented Database System - The story of O2", F.Bancilhon, C.Delobel, P.Kanellakis (Eds.); Morgan Kaufmann, 1992.