C. Beeri, A. Formica, M. Missikoff

INHERITANCE HIERARCHY DESIGN IN
OBJECT-ORIENTED DATABASES

R. 468 Settembre 1998

Catriel Beeri – Hebrew University, Israel. Email: beeri@cs.huji.ac.il.

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.
Abstract

Object-oriented database schema design is still mostly an art. One of the difficulties encountered during design is typing conflicts induced by multiple inheritance. In this paper, we propose a method for treating such a kind of conflicts. Our approach to inheritance conflict solving consists of several ingredients. First, we rely on branding, to introduce ‘type equivalence by name’, thus allowing a designer to distinguish between structurally similar but semantically different types. However, we offer a heuristic that does not require a designer to explicitly state branding declarations. Second, we describe various kinds of conflicts, and we offer a set of procedures that analyze a schema to discover such conflicts, classify them, offering potential solutions, if possible. The procedures have been conceived for a design system that allows a designer maximum flexibility, while guiding him/her to a correct design.
1. Introduction

Inheritance hierarchies represent a key feature in object-oriented systems [24]. An inheritance hierarchy has an engineering import, since it allows more compact code to be developed and a higher degree of reuse to be achieved, and a conceptual import, since it allows a set of definitions (that, generally, in a database schema represent concepts of the application domain) to be organized according to a generalization/specialization relationship. In this paper we focus on object-oriented databases (OODBs) [8], and in particular on schema design activities.

According to ODMG [14], which is the object-oriented data model of reference for our work, an OODB schema consists of a collection of object types (or interfaces), each of which has the form of a labeled set of typed properties (tuple). The properties can be given explicitly or implicitly. In the latter case, the type inherits (part of) its definition from one or more other types (that represent more general entities). Therefore, a schema, that represents the database intent, can be defined according to an inheritance hierarchy incrementally, that is, starting from the most general types and then introducing in the subtypes only the additional properties. The set of objects defined according to a given type represents the extent of such a type.

Inheritance, that allows the designer to define an OODB schema in an incremental way, is in general connected to the notion of type specialization (see, for instance, [34]). A given type can be specialized by introducing additional features (properties or operations) or redefining an inherited feature, according to specific rules (we will elaborate more on this, which is the core of our work). The latter mechanism is often referred to as overriding. In this paper, we address OODB schema design, where data modeling is the focus and functions are not (yet) considered. Therefore, the method proposed in this paper deals with property inheritance.

Rich OODB models allow a specialized type to inherit from more than one supertype. This powerful mechanism, referred to as multiple inheritance, gives more flexibility to the designer, but introduces the risk of inheritance conflicts [6] (or name clashes, according to ODMG terminology). An inheritance conflict takes place if a specialized type inherits the same property, possibly differently typed, from two (or more) supertypes. Since properties cannot appear more than once in a type definition, a multiple occurrence must be reduced to a single property, suitably typed. Inheritance conflict solving consists in finding the ”suitable type” for conflicting properties, or certifying that such a type does not exist, and therefore that there is a contradiction in the schema.

ODMG allows multiple inheritance to be defined, but leaves the treatment of inheritance conflicts open. A correct processing of inheritance conflicts requires a better understanding of the underlying OODB model and a few assumptions at a semantic level, concerning interface extents.

In this work, we propose a technique aimed at processing OODB schemas, identifying inheritance conflicts and suggesting, for certain cases, how to solve them. To this end, inheritance conflicts will be classified, essentially, as amendable or unamendable. Intuitively, the former are inconsistencies for which it is possible to derive a “natural” solution, since the inconsistency is caused by missing information. The proposed method will give the database designer suggestions about the missing information that needs to be added to the schema, without modifying the existing interface definitions. In the case of unamendable conflicts, the method is able to produce a precise diagnosis and pinpoint the interface(s) to be corrected. In this case, no correction will be suggested, since the solution depends on the designer’s view of the application domain. However, in both cases, the final decision is left to the database designer. Finally, once all inheritance conflicts (if there are any) have been removed, the schema is transformed into
a form where all the interfaces exhibit their structures explicitly (i.e., they have the form of a tuple).

Another approach to inheritance conflicts is to allow a class to inherit both attributes, renaming one or both, and letting the user specify which one is inherited from which parent class. Although it may be useful in some situations, we believe that it is not a good approach in a database context. In any case, this approach is not considered in this paper.

1.1. Related work

Inheritance hierarchies (often referred to as ISA-hierarchies) have attracted much attention from both Artificial Intelligence (AI) and Computer Science, in particular within the fields of Programming Languages (PLs), Databases (DBs), and Object-Oriented Analysis and Design (OOA&D). The two key notions of ISA-hierarchies are type specialization (also referred to as subsumption in AI) and inheritance. Within AI, significant work was carried out, in the late 70s, in the context of Semantic Nets [10]. In the 80s, a significant amount of research was developed in studying terminological languages (also referred to as concept languages) and taxonomic reasoning (see, for instance [27]). In this area, the main problem was checking subsumption. The subsumption relation, whose notion was systematically addressed in the FL knowledge representation language [11], states that a concept (an object type) c is a specialization of a concept $c'$ if and only if any instance of $c$ is also an instance of $c'$. Terminological languages are based on the notions of primitive and defined concepts. The definition of a primitive concept specifies only the necessary conditions that an instance has to satisfy (to be a legal instance of such a concept), while in the case of a defined concept, the necessary and sufficient conditions are specified. Subsumption for primitive concepts is somehow axiomatically given, while for defined concepts it can be derived by performing a syntactic comparison of concept definitions. In particular, much of the research has focused on the computational complexity of subsumption for languages with different expressive power (see for instance [16, 27]).

In the field of PLs, the notion of subtyping was introduced mainly for type-checking purposes [13]. In this context, the subtyping relation is derivable according to a pure structural approach, very similar to the computation of subsumption for defined concepts in AI.

Successively, in both the PLs and AI fields, the analysis has been extended to recursive definitions, for which the subtyping relation has been deeply analysed in [1, 2], and whose formal semantics has been extensively investigated in [28].

Within DBs, the ISA-hierarchies were first introduced in semantic data models, developed to provide more powerful abstractions for the specification of database schemas [22, 31]. In the Entity-Relationship model [15], even though generally considered one of the first semantic data model, ISA-hierarchies were introduced only later by [7, 32], in an informal way. Successively, the formal framework proposed in [13] was widely accepted by the DB community, in particular for OODBs, and was adopted by some commercial OODB systems, such as O2 [23].

All methods for OOA&D for Information Systems [9, 17, 30, 25] include the notion of an inheritance hierarchy as a fundamental modeling construct (according to the Object-Oriented paradigm). In this field, inheritance hierarchies are mainly treated in an informal way, allowing the designer to better organize the static information. A constraint often introduced concerns the extent of subtypes (or subclasses) that can have common objects (overlapping inheritance) or not (disjoint inheritance). A formal treatment of inheritance hierarchies is given in [20]. This work, developed according to the Object-Role Modeling techniques (see, for instance, NIAM, PSM
As already mentioned, inheritance and type specialization are two different notions, although closely related. The former is a mechanism that allows a type to inherit (part of) its definition from its supertypes, while the latter allows us to establish a specialization relation between types. Due to the dual import of an inheritance hierarchy, it is a desirable property that, once inheritance has been performed, type specialization among a type and its supertypes holds (also referred to as consistency of a class hierarchy [34]). In PL’s, one often identifies type specialization with subtyping, and then this property is related to the well-known principle of substitutability [24], i.e., the possibility of using an instance of a subtype in a context where an instance of its supertype is expected. However, in the literature less attention has been paid to the issue of how to ensure that, once applied the inheritance process, the result is a consistent schema, especially in the presence of conflicts. An interesting approach for dealing with inheritance and taxonomic information in Logic Programming is proposed by Ait-Kaci [4]. In this approach, inheritance conflicts are solved according to a lattice of type labels representing a signature for inheritance. We borrow from this approach in this work.

Summarizing, the surveyed works address type specialization or inheritance separately, although the above mentioned properties of consistency of class hierarchies and substitutability are recognized as significant requirements. In the next subsection, two possible approaches to inheritance and type specialization are informally illustrated: one is based on a pure structural approach, on the lines of [13], the other is based on a declared hierarchy of type labels, on the lines of [4] (in this paper referred to as naming approach). The different approaches, structural and naming, also have a considerable impact on the computational complexity of inheritance. In fact, following the approach of [4], inheritance can be performed in linear time, while a pure structural inheritance leads to an exponential worst case [18] (however, with regard to consistency checking of ISA-hierarchies, an efficient method that follows the structural approach has been proposed in [19]).

1.2. Two approaches to inheritance and type specialization

The type specialization relation (for tuples and object types), denoted here by (\( <: \)), relies on the basic type specialization golden rule:

"\( \tau_i \) is a specialized type (\( <: \)) of \( \tau_j \) if and only if the former owns at least all the properties of the latter, possibly exhibiting additional ones. Furthermore, in the case of common properties, each occurrence in \( \tau_i \) is typed by a specialized type of the corresponding occurrence in \( \tau_j \)." 

With regard to inheritance, as already mentioned, it is desirable that the consistency of hierarchies and substitutability are guaranteed. Let us consider the examples below (the syntax is taken from [13]).

Example 1.1. In the following schema:

```plaintext
student = {name : string, age : integer, college : string}
employee = {name : string, age : posReal, salary : integer}
empStud = student and employee.
```
while the inheritance conflict due to the double occurrence of the name property is trivially solved, solving the conflict caused by the age property is not straightforward. However, if we assume the existence of a signature for inheritance (i.e., a lattice of type labels), where the type natural is the greatest lower bound, according to the given inheritance relation, of integer and posReal, this conflict can be easily solved as follows:

\[
\text{empStud} = \{\text{name} : \text{string}, \text{age} : \text{natural}, \text{college} : \text{string}, \text{salary} : \text{integer}\}
\]

A different situation arises for inheritance conflicts involving object types (i.e., interfaces). In this case, the greatest lower bound, necessary to solve the conflict, is an object type.

**Example 1.2.** Consider the following schema:

\[
\begin{align*}
\text{student} &= \{\text{name} : \text{string}, \text{vehicle} : \text{bike}, \text{college} : \text{string}\} \\
\text{employee} &= \{\text{name} : \text{string}, \text{vehicle} : \text{motor_bike}, \text{salary} : \text{integer}\} \\
\text{bike} &= \{\text{maker} : \text{string}, \text{color} : \text{string}\} \\
\text{motor_bike} &= \{\text{maker} : \text{string}, \text{color} : \text{string}, \text{power} : \text{string}\} \\
\text{work_stud} &= \text{student} \text{ and } \text{employee}
\end{align*}
\]

According to the naming approach, that relies on a pre-specified hierarchy on the type names, the \text{work_stud} interface contains a contradiction that arises in inheriting the vehicle property, since there is no given precedence between the conflicting interfaces motor_bike and bike. On the other hand, according to the structural approach, by applying the type specialization golden rule, it can be derived that motor_bike <: bike, therefore the schema of the Example 1.2 does not contain any contradiction and the \text{work_stud} interface can be rewritten as follows:

\[
\text{work_stud} = \{\text{name} : \text{string}, \text{vehicle} : \text{motor_bike}, \text{college} : \text{string}, \text{salary} : \text{integer}\}
\]

As seen in the previous example, the structure of types often reflects the correct type specialization relationship, without requiring a designer to explicitly specify a hierarchy on the type names. This is not always the case. To better clarify the point, let us consider the following interfaces:

\[
\begin{align*}
\text{person} &= \{\text{name} : \text{string}, \text{age} : \text{integer}\} \\
\text{student} &= \{\text{name} : \text{string}, \text{age} : \text{integer}, \text{college} : \text{string}\} \\
\text{dog} &= \{\text{name} : \text{string}, \text{age} : \text{integer}, \text{breed} : \text{string}\}
\end{align*}
\]

According to the structural approach, we can derive:

\[
\text{student} <: \text{person}, \text{dog} <: \text{person},
\]

while according to the naming approach, these three types are not in a type specialization relation, since no relation between them is given. Quite clearly, we would like to consider \text{student}, but not \text{dog}, as a specialized type of \text{person}. This can be achieved by providing an explicit hierarchy on these type names that reflects the desired relations. One of our contributions in this work is to show how such a hierarchy can be derived without imposing additional burdens on a schema designer. Specifically, in this paper, we present a solution that merges the two approaches: in the proposed method, inheritance is performed taking into account the structures of the interfaces (i.e., the sets of typed properties), together with a partial order over interface
names, defined according to the inheritance hierarchy declared in the schema. Type specialization between interfaces requires, therefore, that the golden rule holds and the interface labels are hierarchically declared in the partially ordered set. This issue is elaborated in Subsection 2.3.

The method proposed in this paper is based on a few assumptions and a formal data model that will be illustrated in the following section. In Section 3, a simple inheritance (or expansion) process is first proposed and, then, the inheritance process is informally introduced by means of a few examples. The algorithm that implements the proposed method is presented in Section 4, while Section 5 ends the paper with the conclusions and a few indications about the future work.

2. Formal Basis

We now present an OODB model and a schema definition language, similar (but not identical) to the ODMG standard. We also discuss the semantics of schemas, when the structure of each object type is explicitly given or, in the presence of inheritance relationships, is made explicit by expanding implicit definitions.

2.1. Object Schemas

Informally, a schema contains definitions for literal types (i.e., types of complex values) and object types. Literal types are constructed from atomic types such as integer, string by constructors, such as tuple or set. Their domains are fixed, essentially predefined. Object types are defined (i.e., user-defined), and their domains are populated by explicit object creation requests. In an OODB, object type definitions are typically included in class definitions, that additionally contain method code and other components. Here, as our concern is schema design, we deal only with the type component. As already mentioned, in this paper, we follow the ODMG terminology and, to distinguish between the two kinds of types, we refer to object types as interfaces. 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), (iv) expressions constructed from (i) and (ii) by set constructor (i.e., multivalued properties). In case (i) and (ii) properties are referred to, in the ODMG standard, as attributes and relationships, respectively [14], but we do not make this distinction here. These can easily be distinguished by their types.

The feature that we are interested in is that of inheritance, by which an interface definition can list other interfaces from which the current interface inherits all or some of its properties.¹ (All, if no properties are explicitly listed for the interface, some otherwise.) For a schema to be well-defined, its inheritance graph must be acyclic, i.e. form a hierarchy. The ODMG allows multiple inheritance and overriding, which, as already mentioned in Section 1, are both sources of inheritance conflicts [6]. The hierarchy is, in general, represented as a DAG, that becomes a tree in the absence of multiple inheritance. Notationally, we follow Cardelli’s proposal [13], using and to express multiple inheritance.

In the following definitions, we assume that countable sets \( T, P \) of interface names and property names, respectively are given.

¹Thus, we consider here structure inheritance, not behaviour inheritance.
Definition 2.1. [Interface definition] An interface definition has the form:

\[ \tau = \tau_1 \text{ and } \ldots \text{ and } \tau_n \text{ and } \{p_1 : \zeta_1, \ldots, p_m : \zeta_m\}, \]

\[ (1) \]

where \( n \geq 0, m \geq 0, \) and in each definition, either \( n > 1, \) or \( m > 0. \)

In (1) the \( \tau_i \)'s are interface names from \( T, \) the \( p_j \) are property names from \( P, \) and each \( \zeta_k \) is an atomic type, an interface name from \( T, \) or a type composed by using type constructors. In this definition, \( \tau \) is defined, and every interface name on the right is used. The part \( '\tau_1 \text{ and } \ldots \text{ and } \tau_n' \) (if \( n > 0 \)) in the definition is called the inheritance component of the definition; the part \( '\{p_1 : \zeta_1, \ldots, p_m : \zeta_m\}' \) (if \( m > 0 \)) is called the explicit component. A definition that contains only an explicit component is explicit, otherwise it is implicit.

If a definition for an interface \( \tau \) has no explicit component, then it must have an inheritance component. In that case, we require that this component contains at least two names, for otherwise the defined interface is identical to the one used to define it.

Definition 2.2. [OODB Schemas] A schema \( \Sigma = (T, P, Def) \), consists of finite set \( T \subset T, P \subset P, \) and a finite set \( Def \) of interface definitions such that the following holds.

- Every name in \( T \) has precisely one definition in \( Def. \)
- Every interface name used in \( Def \) is in \( T. \)
- Let \( directDesc \) (directDescendant) be the binary relation on \( T \) defined by:

\[ directDesc(\tau, \tau_i), \text{ if } \tau_i \text{ is used in the inheritance } \]

component of the definition of \( \tau. \)

Then if \( directDesc(\tau, \tau_i) \) holds, there is no chain of length two or more, of \( directDesc \) relations from \( \tau \) to \( \tau_i. \)

- Let \( descOf \) (descendantOf) be the reflexive and transitive closure of \( directDesc. \) Then \( descOf \) is acyclic, i.e., a partial order.

The first two requirements are obvious: to prevent ambiguity, an interface should have a unique definition, and clearly every interface \( \sigma \) that is used in the definition of some interface \( \tau \) must itself be defined in the schema. The acyclicity requirement for \( descOf \) is well-known; it is necessary to support the semantics of definitions with inheritance components as inheritance of structure. The constraint on \( directDesc \) is also quite natural: if \( \sigma_1 \) inherits from \( \sigma_2 \) that inherits from \( \sigma_3, \) there is no point to also state that \( \sigma_1 \) inherits from \( \sigma_3. \)

In the sequel, whenever \( \leq \) denotes a partial order, we use \( s < t \) to denote that \( s \leq t \) and \( s \neq t. \)

Definition 2.3. [Reduct] Given a poset \( (S, \leq), \) the reduction of \( \leq \) is denoted by \( reduct(\leq). \)

That is, \( s \text{ reduct}(\leq) t \) holds if and only if \( s < t \) holds, and for no \( u \) it is the case that \( s < u < t. \)

In terms of \( reduct, \) the requirement from \( directDesc \) can now be stated as:

\[ directDesc = reduct(directDesc) = reduct(descOf). \]

In the sequel, schemas will be presented as a set of definitions \( Def, \) derivation of \( T, P \) from the definitions is trivial.
Example 2.1. Consider the definitions:

\[
\text{student} = \{ \text{name} : \text{string}, \text{friend} : \text{student}, \text{college} : \text{string} \} \\
\text{employee} = \{ \text{name} : \text{string}, \text{friend} : \text{employee}, \text{salary} : \text{integer} \} \\
\text{work} = \text{student} \text{ and employee}.
\]

It is easy to verify that it is a schema. In this schema, \text{employee}, \text{student} have explicit definitions, i.e., their definitions do not use inheritance, while \text{work} inherits its structure from the other two.

Definition 2.4. [Explicit schemas] A schema is in \textit{explicit form}, denoted \textit{EF}, if all its definitions are explicit; otherwise it is in \textit{implicit form}, denoted \textit{IF}. Given a schema in \textit{EF}, for each interface \(\tau\), if its definition is \(\tau = \{p_i : \zeta_i\}_{1 \leq i \leq n}\), then we denote the type on the right side by \(i(\tau)\). (\(i\) stands for the interface schema, or \textit{i-schema} for short.)

For instance, the above schema is not in \textit{EF}, because it contains one implicit definition.

Given a schema with implicit definitions, we would like to expand it to an explicit schema, and in the process discover if the original definitions contain potential inconsistencies. But, note that the inheritance relationship \textit{descOf}, as defined above, is derived from the inheritance components of definitions. As these are processed to obtain explicit definitions, the inheritance information they contain is lost, and indeed in an explicit schema the derived \textit{descOf} is empty. Since, we do want to keep a record of the inheritance relationship even after expansion, we need to keep such a record as an explicit independent component of the schema. Hence, we define:

Definition 2.5. [Full explicit schema] A \textit{full explicit schema} \(S\) is a pair \(S = (\Sigma, \text{descOf}_S)\), where \(\Sigma = (T, P, \text{Def})\) is a schema in \textit{EF}, and \textit{descOf}_S is a (reflexive) partial order on \(T\), called the \textit{inheritance relation} of the schema.

Similarly, if \(\Sigma\) is a schema in \textit{IF}, we will denote the relation \textit{descOf}, derived from its definitions, by \textit{descOf}_\Sigma. We expect that if such a schema is successfully expanded to an \textit{EF} schema \(S\), then this relation will be its inheritance relation \textit{descOf}_S. This is indeed the case for our expansion algorithm in Section 4. Of course, \textit{descOf} in an implicit schema is represented by \textit{directDesc}.

Note that the definition above does not require that if \(\tau_1 \text{ descOf } \tau_2\) then the interface of \(\tau_1\) includes all the components of the interface of \(\tau_2\). This is treated below.

2.2. The Semantics of Schemas

Before considering the expansion process, we need to understand the semantics of schemas. We follow the definitions of [3]. In the following, let \(S = (\Sigma, \text{descOf}_S)\) be a full explicit schema.

Definition 2.6. [Disjoint oid assignment] Let \(O\) be a countable set of (abstract) object identifiers (oid’s), that is disjoint from the domains of all atomic types. An \textit{oid assignment} for a schema \(\Sigma\) in \textit{EF} is a function \(\pi\) that assigns to each interface name in \(T\) a finite subset of \(O\), such that for \(\sigma, \tau \in T, \sigma \neq \tau, \pi(\sigma) \cap \pi(\tau) = \emptyset\).

Note that, according to [13], in our approach an oid is a partial function over property names. Now, we can associate domains with the types of a schema. The domains for tuple and set constructions are defined as usual. The only addition required here is to define the domains for the interfaces.
Definition 2.7. [The over set] The set of types over a set of interface names \( T \) and a set of property names \( P \) is the smallest set that contains the atomic types, the set \( T \), and is closed under tuple and set formation. For a schema \( \Sigma = (T, P, \text{Def}) \), we refer to the types over \( T \) and \( P \) also as the types over \( \Sigma \).

Definition 2.8. [Domains] Let \( \Sigma \) be a schema in \( EF \), and \( \pi \) be an oid assignment. The domains for the types over \( \Sigma \) are defined as follows. The domain \( D(b) \), for an atomic type \( b \) is given. For \( \tau \in T \), its domain \( D(\tau) \) is \( \pi(\tau) \). The domains for tuple and set types are constructed as usual: \( D(\{p_i : \zeta_i \}_{1 \leq i \leq n}) \) is the set of functions \( \rho \) on \( \{p_1, \ldots, p_n\} \), such that \( \rho(p_i) \in D(\zeta_i) \), and \( D(\{\tau\}) \) is the collection of all finite sets of elements of \( D(\tau) \).

Now, we can introduce the next component, namely the connection between objects and their states.

Definition 2.9. [State function, database] Given \( \Sigma, \pi \) as above, a state function \( \nu \) is a function that assigns to each \( \rho \) in \( D(\tau) \), where \( \tau \) is an interface type, the record \( \{p_i : \rho(p_i)\} \), where \( \rho(p_i) \in D(\zeta_i) \), and \( i(\tau) = \{p_i : \zeta_i\} \). The pair \( (\pi, \nu) \) is a database, and is called an instance of \( \Sigma \).

The above definition does not take into consideration the inheritance relationship. While the object sets corresponding to different interfaces are required to have no objects in common (see Def. 2.6), the usual approach is to define the extent of a class to include the extents of its subclasses. That is, the extent of an interface is the union of its set of objects, and those of the interfaces that are ‘under’ it in \( \text{descOf} \). It follows that when users query or otherwise manipulate the extent of an interface, objects of descendant interfaces are included. But note that these extents are heterogeneous sets, as they contain objects with different structures. Clearly, this heterogeneity must be restricted to guarantee that such heterogeneous sets have a well-defined meaning to the users. Furthermore, two sibling classes having common objects require the presence of a common subclass. In this way, disjoint oid assignment does not preclude overlapping inheritance to be achieved.

Following [3, 13], we define:

Definition 2.10. [Specialized type relation] For a full explicit schema \( S \) as above, for types over \( \Sigma \), the specialized type relation, denoted \(<:\), is defined as the smallest relation such that:

- All types: \( \tau <: \tau \).
- Interfaces: \( \tau_1 <: \tau_2 \) if \( \text{descOf}_S(\tau_1, \tau_2) \).
- Tuples: \( \{p_i : \zeta_i\}_{1 \leq i \leq n+m} <: \{p_i : \eta_i\}_{1 \leq i \leq n} \) if \( \zeta_i <: \eta_i \) for \( 1 \leq i \leq n \).
- Sets: \( \{\tau_1\} <: \{\tau_2\} \) if \( \tau_1 <: \tau_2 \).

A related requirement is that compile-time type checking can guarantee that manipulations of objects via methods is always well-defined. This is often expressed by a subtype relation among types, that satisfies the substitutability property:

Wherever an instance of a type is expected, an instance of a subtype can also be used.
In our context, using an object (an element in some \( \pi(\tau) \)) implies accessing its state. Hence, the property needs to be guaranteed for the tuple types associated with the interfaces. Note that our definition of specialized type allows changing the types of properties in tuple types in a covariant fashion. This coincides with the definition of subtyping for *queries*. That is, if in a schema whenever a type \( \tau \) inherits from a type \( \tau' \) then \( \tau <: \tau' \), then any (read-only) function that passes compile-time type checking, that expects objects of type \( \tau' \), can work with no change on objects of \( \tau \). This does not hold, however, for updates, for which subtyping requires contravariant changes of property types [12], while we allow covariant changes. Since updates are required in the general use of an OODB, our specialized type relation is different from subtyping and does not guarantee the absence of run-time type errors; run-time checks are needed. Nevertheless, it is commonly accepted that covariant changes reflect real-world situations, and should be supported even at the expense of costlier run-time consistency checks. An analytic approach that discovers where run-time checks are needed is described in [5]. We follow this approach and allow covariant changes. Thus, we define well-formed schemas as follows:

**Definition 2.11. [Well-formed schemas]** A full explicit schema \( S \) is well-formed if for all types \( \tau_1, \tau_2 \in T \), if \( \text{descOf}_S(\tau_1, \tau_2) \) then \( i(\tau_1) <: i(\tau_2) \).

### 2.3. Type specialization by name

Our goal now is to consider implicit schemas, and consider the obstacles one may encounter when trying to convert them to fully explicit schemas. Before that, however, we need to consider one last issue. The definition of type specialization given above is *structural*, that is, it only relies on the structure of types. Even for fully explicit schemas, this may be misleading.

**Example 2.2.** Consider the schema:

\[
\text{person} = \{\text{name} : \text{string}, \text{age} : \text{integer}, \text{address} : \text{string}\} \\
\text{dog} = \{\text{name} : \text{string}, \text{age} : \text{integer}\}
\]

While the fact that the type checking mechanism may allow types that should be incomparable to be mixed in expressions may seem a small problem for explicit schemas (in the above example we derive \( \text{person} <: \text{dog} \)), this kind of confusion is a more serious obstacle when an implicit schema needs to be expanded. A mechanism to differentiate between types with similar structures is well-known and used in many programming languages, namely to associate names with, or to *brand*, the types. For example, in C, a Struct has a name, and Structs with different names are different types. In Logic Programming, e.g. in Prolog, function symbols are essentially named constructors. This idea was extended to subtyping by Ait-Kaci [4]. To incorporate the idea into our framework, we need a few extensions.

**Definition 2.12. [Branded tuple constructor]** A *branded* tuple constructor has the form \( \{N\} \). The domain of a type \( \{N \ p_i : \tau_i\}_{1 \leq i \leq n} \) is the set of functions \( \rho \) on \( \{N, p_1, \ldots, p_n\} \) such that \( \rho(N) = N \) and \( \rho(p_i) \in D(\tau_i) \).

In order to use branding, we need to assume that the tuple types used in interface definitions are branded. But, we do not expect schema designers to directly use this mechanism. Rather, we assume their intentions are reflected in the interface names and in the inheritance relationships they use. Hence, we use interface names as brands, and introduce them automatically into the definitions.
Definition 2.13. [Branded schemas] For a schema \( \Sigma \), in implicit or (full) explicit form, the branded version of \( \Sigma \) is obtained by replacing each definition of the form (1) in Def. 2.1 above by:

\[
\tau = \tau_1 \text{ and } \ldots \text{ and } \tau_n \text{ and } \{ \tau_{p_1 : \zeta_1, \ldots, p_m : \zeta_m} \}, \ n \geq 0, m \geq 0 \quad (2)
\]

Clearly, this implicit branding solves the problem illustrated in the example above, but now we need to reconsider type specialization. Since the brands for different interfaces are different, a full explicit branded schema cannot be well-formed. To address this issue, we need to incorporate into the type specialization relationship a partial order on brands. Note that now we need a different notion of types over a set of interfaces.

Definition 2.14. [Branded over set] Let \( T \) be a set of interface names, and \( P \) be a set of property names. The set of branded types over \( T \) and \( P \) is defined as the set of types over \( T \) and \( P \), except that tuple types are constructed only with branded tuple constructors, where the set of brands is \( T \).

Definition 2.15. [Branded specialized type relation] For a full explicit schema \( S \) as above, the specialized type relation on the branded types over \( \Sigma \) is defined as in Def. 2.10, except for tuple types:

- Tuples: \( \{ \tau_{p_1 : \zeta_1, \ldots, p_m : \zeta_m} \} \) if \( \text{descOf}_{S}(\tau, \sigma) \) and \( \zeta_i <: \eta_i \) for \( 1 \leq i \leq n \).

Now we can restate the definition of well-formedness:

Definition 2.16. [Well-formed schemas] A full explicit branded schema \( S \) is well-formed if for all types \( \tau_1, \tau_2 \in T \), if \( \text{descOf}_{S}(\tau_1, \tau_2) \) then \( i(\tau_1) <: i(\tau_2) \).

3. The Simple Expansion Process

In the remainder of this paper, we present an algorithm for the expansion of implicit schemas, whose interfaces have properties typed with atomic types or interface labels. In particular, in this section, we present a simple algorithm that tries to expand a given implicit schema to a full explicit one. We formalize the relationship between the two schemas when the algorithm succeeds. However, the algorithm may fail. In the rest of the section we illustrate via examples situations where it fails, and our proposed solution.

Given an interface definition \( \tau \) in implicit form, the expansion, i.e., the inheritance process, consists essentially of removing the and construct and inheriting the typed properties of the interfaces used in the inheritance component of \( \tau \). As already mentioned, multiple inheritance and overriding may generate inheritance conflicts. In our case, these manifest as multiple types for a property. The process succeeds if there are no conflicts. Our goal is to identify various kinds of conflicts, and offer appropriate solutions for each kind.

3.1. Simple Expansion

The following concepts are used in the expansion algorithm.
Definition 3.1. [Maximal/greatest lower bounds] Let \((T, \leq)\) be a poset. For a subset \(S \subseteq T\), and an element \(t \in T\), we write \(t \leq S\) if \(t \leq s\) for all \(s \in S\). An element \(t\) is a maximal lower bound of \(S\), denoted \(mlb(S)\), if \(t \leq S\) holds, and there is no \(t'\) such that \(t < t' \leq S\). It is a greatest lower bound of \(S\), denoted \(glb(S)\) if it is an \(mlb\) of \(S\), and further, for any \(t'\), if \(t' \leq S\) holds then \(t' \leq t\).

Note that \(s = glb(S)\) if and only if it is the unique \(mlb\) of \(S\). One direction (\(glb\) implies unique \(mlb\)) is obvious. For the other direction, assume \(t\) is the unique \(mlb\) of \(S\). Note that given \(t' \leq S\), if it is not an \(mlb\) of \(S\), then for some \(t''\) we have that \(t' < t'' \leq S\). By induction it follows that if \(t' \leq S\) then it is either an \(mlb\) or \(t' < t\), where \(t\) is an \(mlb\). Since \(t\) is the unique \(mlb\), it follows that if \(t' \leq S\) then \(t' = t\) or \(t'' < t\). The claim follows.

The definitions apply, naturally, to the partial order on schemas:

Definition 3.2. [Maximal/greatest lower interfaces] Given a branded schema \(\Sigma\) as above, for any set of interfaces \(S \subseteq T\), an interface \(\sigma\) is a maximal/greatest lower interface of \(S\), denoted \(mli(S)/gli(S)\), respectively, if it is a maximal/greatest lower bound of \(S\) in \((T, \text{descOf})\).

Next, we consider inheritance of property types. (Type here means both an interface and an atomic type.)

Definition 3.3. [Conflict Set] Let \(\Sigma\) be an implicit schema. For each property \(p\) used in it, and each defined interface \(\tau\), the set of types of \(p\) for \(\tau\), denoted \(\text{typesOf}(\tau, p)\), is defined as follows. Assume the interface definition for \(\tau\):

\[
\tau = \tau_1 \text{ and } \tau_2 \ldots \text{ and } \tau_n \text{ and } \{p_1 : \zeta_1, \ldots, p_m : \zeta_m\}, \ n \geq 0, \ m \geq 0
\]

Then, \(\text{typesOf}(\tau, p) = \bigcup_k^n \text{typesOf}(\tau_k, p) \cup \text{type}_\tau(p)\), where \(\text{type}_\tau(p)\) is \(\{\zeta_k\}\) if \(p = p_k\) for some \(k\), and is empty otherwise. We refer to \(\text{typesOf}(\tau, p)\) as the conflict set for \(p\) in \(\tau\).

Note that had we tried to handle all kinds of inheritance conflicts, then we should have defined \(\text{typesOf}\) as a bag, thus reflecting for each type the number of times it is inherited. It is well known that even in the case that the property with the same type is inherited from several sources, a conflict may occur, and in some cases splitting the property into two by renaming is advocated. However, in this paper this issue is not treated. (A treatment can be incorporated as a preprocessing step to our algorithm.) That is, we only consider the conflicts where a property inherits more than one type. Thus, sets suffice for our needs.

Clearly, the sets \(\text{typesOf}(\tau, p)\) can be computed by a recursive procedure that starts at the leaves of the inheritance digraph, i.e., the interfaces that have no \(\text{descOf}\) parents. If the set is empty, then that interface does not have the property in its set of properties. If the set has size one, than there is no conflict (or one may say the conflict is trivial). This leads to the expansion algorithm in Figure 1.

For the case where this simple algorithm succeeds, we can easily prove the following.

Proposition 3.1. If the algorithm SimpleExpand succeeds, then the resulting schema is well-formed.
SimpleExpand  // input: a schema $\Sigma$ in implicit form
// output: a full explicit schema $S$, or 'fail'

compute $\text{typesOf}(\tau, p)$, for all $\tau, p$ in $\Sigma$
if $|\text{typesOf}(\tau, p)| > 1$, for some $\tau, p$ in $\Sigma$
    return 'fail'
else,
    let $S = (\text{Defs}rolley, \text{descOf}rolley)$ be the schema where, for each $\tau$ of $\Sigma$:
        $\tau$ is in $\text{Defs}rolley$ with the $i$-schema, $i(\tau)$, s.t.:
            $p$ is in $i(\tau)$ iff $\exists \sigma \in \text{typesOf}(\tau, p)$
            and $p$ is typed with $\sigma$
        $\text{descOf}rolley = \text{descOf}Sigma$
    return $S$

Figure 1: A Simple expansion algorithm

3.2. Examples

The IF schema of Example 2.1 contains two inheritance conflicts. The first, regarding the name property, is trivial, while the second, regarding the friend property, is not. The latter is solvable, however, since work_stud is a gli of student and employee, according to the descOf relation:

$\text{descOf(work_stud, student)}$
$\text{descOf(work_stud, employee)}$.

Therefore, the inheritance process can succeed by choosing this gli as the type for the property, returning the following full EF schema:

$\text{Def:}$

$\text{student} = \{\text{name:} \text{string}, \text{friend: student, college:} \text{string}\}$
$\text{employee} = \{\text{name:} \text{string}, \text{friend: employee, salary:} \text{integer}\}$
$\text{work_stud} = \{\text{name:} \text{string}, \text{friend: work_stud, college:} \text{string, salary:} \text{integer}\}$

whose associated directDesc relation is graphically represented in Figure a.
Example 3.1. Consider now the following schema:

\[
\text{student} = \{\text{name} : \text{string}, \text{vehicle} : \text{push\_bike}, \text{college} : \text{string}\}
\]

\[
\text{employee} = \{\text{name} : \text{string}, \text{vehicle} : \text{motor\_bike}, \text{salary} : \text{integer}\}
\]

\[
\text{motor\_bike} = \{\text{maker} : \text{string}, \text{power} : \text{string}\}
\]

\[
\text{push\_bike} = \{\text{maker} : \text{string}, \text{speed} : \text{string}\}
\]

\[
\text{work\_stud} = \text{student} \text{ and } \text{employee}
\]

whose associated descOf is the same of the Example 2.1. In this case, the inheritance conflict due to the vehicle property is not directly solvable (there is no mli for the interface names motor\_bike and push\_bike), i.e., this schema contains an inconsistency.

However, within the inconsistencies, we distinguish the ones due to inheritance conflicts involving only interfaces from the ones involving at least one atomic type. In fact, for the former, the introduction of a new interface or a simple modification of the descOf relation allow the solution of the conflict, while, for the latter, a modification of the existing definitions of the schema is required. For instance, in the above example, we say that the inheritance conflict due to the vehicle property is amendable since it could be solved by simply introducing a new interface in the schema whose name is, for instance, push\&motor\_bike, that inherits from both the conflicting interfaces motor\_bike and push\_bike. This modification allows the inheritance process to succeed, and the following full EF schema is obtained:

\[
\text{Def:}
\]

\[
\text{student} = \{\text{name} : \text{string}, \text{vehicle} : \text{push\_bike}, \text{college} : \text{string}\}
\]

\[
\text{employee} = \{\text{name} : \text{string}, \text{vehicle} : \text{motor\_bike}, \text{salary} : \text{integer}\}
\]

\[
\text{motor\_bike} = \{\text{maker} : \text{string}, \text{power} : \text{string}\}
\]

\[
\text{push\_bike} = \{\text{maker} : \text{string}, \text{speed} : \text{string}\}
\]

\[
\text{work\_stud} = \{\text{name} : \text{string}, \text{vehicle} : \text{push\&motor\_bike}, \text{college} : \text{string}, \text{salary} : \text{integer}\}
\]

\[
\text{push\&motor\_bike} = \{\text{maker} : \text{string}, \text{power} : \text{string}, \text{speed} : \text{string}\}
\]

whose direct\_Desc relation, graphically represented in Figure b, is an extension of the direct\_Desc relation derived from the original schema.

![Figure b](image-url)

It is easy to verify that the full EF schema above is well-formed (since type specialization holds for interfaces in descOf.)
Another case that can be treated is when a *typesOf* set has more than one mli. In this case, a sensible approach is to factor out common properties into a new interface whose position is below the set of interfaces in the *typesOf* set, and above their multiple mli’s. In some cases, this approach introduces an interface that has the same properties and types as an existing one. In many cases, the two should be merged, but the final word is of course the designer’s. The next two examples illustrate this case.

**Example 3.2.** Consider the following schema.

```plaintext
student = {name : string, friend : student, college : string}
employee = {name : string, friend : employee, salary : integer}
sport_man = {name : string, sport : string, salary : integer}
work_stud = student and employee

The inheritance relation associated with this schema is graphically represented in Figure c.
```

In this case, for the student and employee interfaces there exist two mli’s, namely, work_stud and tennis_player. Factoring out the common properties, we obtain a suggestion for a new interface, that is identical in structure to work_stud. Therefore, the designer is suggested to modify the schema as follows:

```
student = {name : string, friend : student, college : string}
employee = {name : string, friend : employee, salary : integer}
sport_man = {name : string, sport : string, salary : integer}
work_stud = student and employee

The new directDesc relation is now represented in Figure d.
```

For this schema, an *EF* schema can be derived using the approach previously illustrated. The inheritance conflict due to the friend property can be uniquely solved by using the work_stud interface as follows:

```
student = {name : string, friend : student, college : string}
employee = {name : string, friend : employee, salary : integer}
sport_man = {name : string, sport : string, salary : integer}
work_stud = {name : string, friend : work_stud, college : string, salary : integer}
tennis_player = {name : string, friend : work_stud, college : string, salary : integer, sport : string}
```

![Figure c](image-url)
A similar situation, where a newly suggested interface has the same \textit{i-schema} as an existing one is illustrated in the Example 1.2, where the \textit{descOf} partial order is again the same shown in Figure a. The difference here is in the position of the interfaces in the \textit{directDesc} hierarchy.

\begin{figure}[h]
\centering
\includegraphics[width=0.5\textwidth]{fig_d.png}
\caption{Figure d}
\end{figure}

In this case, the system first attempts to solve the amendable inheritance conflict related to the \textit{vehicle} property by the introduction of a new interface. But, since the \textit{i-schema} of the new interface coincides with the one of the \textit{motor.bike} interface, the use of the already existing interface \textit{motor.bike} is suggested to the designer, rather than a new one. The explicit definition of \textit{work.stud} becomes:

\[
\text{work.stud} = \{\text{name : string, vehicle : motor.bike, college : string, salary : integer}\}
\]

and the inheritance relation associated with the resulting \textit{EF} schema is modified by adding the pair \textit{directDesc(motor.bike, bike)}, as shown in Figure e.

\begin{figure}[h]
\centering
\includegraphics[width=0.5\textwidth]{fig_e.png}
\caption{Figure e}
\end{figure}

However, note that, in both the above examples, it is possible to realize that the definition of the newly introduced interface will coincide with that of an already existing interface in the schema, by simply checking the set inclusion relation between the components of the involved interface definitions (in particular, the set inclusion between the inheritance components of \textit{work.stud} and \textit{tennis.player}, in the Example 3.2, and the typed properties of the definitions of \textit{bike} and \textit{motor.bike}, in the Example 1.2). In Section 4, we will see that this check is performed by the \textit{Restructure} procedure.
Example 3.3. Finally, consider the schema:

\[
\begin{align*}
\text{student} &= \{ \text{name} : \text{string}, \text{phone} : \text{string}, \text{college} : \text{string} \} \\
\text{employee} &= \{ \text{name} : \text{string}, \text{phone} : \text{integer}, \text{salary} : \text{integer} \} \\
\text{work} &\cup \text{stud} = \text{student} \cup \text{employee}
\end{align*}
\]

It is easy to verify that this schema, whose \( \text{descOf} \) partial order coincides with the one of the Example 2.1, contains an inconsistency in correspondence to the \text{phone} property. Such a conflict is unamendable (the conflicting types, \text{string} and \text{integer}, are atomic) and, therefore, no correction is possible without modifying an existing interface definition.

4. The Expansion Process

This section presents the inheritance process, aimed at transforming an \( \text{IF} \) schema into a corresponding \( \text{EF} \) schema. The process follows the ideas illustrated via examples in the previous section. The problem that needs to be addressed is the possibility that for some object property \( p \), the set \( \text{typesOf}(\tau, p) \) has more than one element. If it has a gli, then we can use this gli as the type of the property, but what should we do if it does not have one?

We recall that for any given partial order \( (T, \leq) \), there is a well-known procedure to enrich it to a partial order with maximal lower bounds (mlb's) and minimal upper bounds (mub's) defined for all subsets. Briefly, it works as follows. For a subset \( S \) of \( T \), let \( l(S) \) denote the set \( \{ t \in T \mid t \leq S \} \), and \( u(S) \) denote the set \( \{ t \in T \mid t \geq S \} \), and denote by \( ul \) the composition of the two mappings, i.e. \( ul(S) = u(l(S)) \). Then for any set \( S \), \( ul(ul(S)) = ul(S) \). In this approach, the completion procedure consists of adding to \( T \) a new element, denoted say by \( ulS \), for each \( ul(S) \). It is known that with these additions, mlb's and mub's exist for all sets, although they are not necessarily unique.

From our viewpoint, however, this procedure has several disadvantages. First, it may introduce an exponential number of new elements. We prefer to introduce new interfaces only if they are needed for providing types to properties. Furthermore, the completion procedure introduces new elements even in cases when existing elements may serve the role of an mlb or mub. We would like to introduce new object types only if existing object types cannot serve the role that is needed. Finally, it can be seen that when \( \text{typesOf}(\tau, p) \) has no mli, then \( l(\text{typeOf}(\tau, p)) \) is empty, hence \( ul(\text{typeOf}(\tau, p)) \) contains all interfaces. Thus, the new interface in this case is a gli of all existing interfaces. This does not seem appropriate for a schema design situation. We prefer in such a case to add an interface to serve as a gli of the set \( \text{typesOf}(\tau, p) \) only.

Our approach is detailed in the following. Its main component is the \textit{Expand} procedure in Figure 2. Before going into the details, we note that our viewpoint is that the designer can interrupt a design process at any stage, to manually edit the schema. Hence our approach is a heuristics-based design process, with a set of tools to assist the designer, rather than an algorithm that automatically solves the conflicts. This is reflected in the procedures as follows:

1. The schema \( \Sigma \) is a global variable, shared by all procedures, and affected by side-effects, that is, updates performed in the procedures. Each change made to it is persistent even if the designer later exits from a procedure. Thus, the designer can interrupt a design session, preserving partial results, and resume it later from the same state, or from another state obtained by manual modification of the schema.

2. The designer can suspend or exit from each of the procedures. In particular, print messages are included to emphasize points where the designer may opt to exit or temporarily suspend...
Expand  // input: a schema Σ in implicit form
  // output: full explicit schema S

| compute typesOf(τ,p), for all τ,p in Σ
| while ∃ τ,p, s.t. | typesOf(τ,p) > 1 and ∅ gli(typesOf(τ,p)),
| do call Solve(typesOf(τ,p)) end   // this call may not return
| let S = (DefS, descOfS) be the schema where, for each τ of Σ:
| τ is in DefS with the i-schema, i(τ), s.t.:
| p is in i(τ) iff | typesOf(τ,p) | ≥ 1
| and p is typed with gli(typesOf(τ,p))
| descOfS = descOfΣ
| Compact(S)  // this call never returns

Figure 2: The Expand algorithm

The process, to perform manual editing. The return value of a procedure is the value it returns upon normal completion.

The nodes of the given implicit schema form a digraph under directDesc. For each interface τ, and for each property p in the schema, the set typesOf(τ,p) is computed. If each of these sets has a gli (being a singleton is a special case), then this gli is the chosen type for the property p at τ, and the full explicit schema is derived (possibly by scanning the nodes from the leaves, such that when a node is processed, all its direct parents essentially have explicit i-schemas). Otherwise, the Solve procedure is called, to search for a solution.

The case where the set has an atomic type as an element can only be corrected by the schema designer, by changing some type definitions, since two different atomic types2, or an atomic type and an interface name can never have a common specialized type (see the Example 3.3). If all the types in typesOf(τ,p) are interface names, then, we first check if one of the interfaces in typesOf(τ,p) may play the role of a gli. This step is performed by the Restructure procedure that is, essentially, an optimization step, i.e., it directly suggests how to modify the directDesc relation, rather than introducing a new interface whose definition coincides with an existing one. Then, if the Restructure succeeds the conflict may have been solved (see, for instance the Example 1.2), otherwise we consider two subcases.

In the first case (amendable conflict), that concerns typesOf(τ,p) sets having no mli, a new interface is introduced, to serve as a gli for it (see the Example 3.1). The set of properties for this new interface is determined by its intended position in the descOf hierarchy, that is, we define it as the and of the interfaces in the typesOf(τ,p) set. (The explicit form for this interface will need to be computed at some later stage.) We can now make this new interface the type of property p at τ.

In the second case (factorable conflict), i.e., typesOf(τ,p) has more than one mli, we have several options: these are handled in the Factor procedure. The first is to make one of the mlis the type of p; this can be the result of an arbitrary choice of the designer, that can be implemented either by adding a definition to the explicit component of τ, or by changing the existing type of p in τ. Another way of achieving this goal is to change the descOf relations between the mlis, by using the Restructure procedure, this time applied to the set of mlis of

---

2This restrictive assumption can be relaxed by introducing a specialization hierarchy also for atomic types.
Solve // input: a conflict set \( C = \text{typesOf}(\tau, p) = \{\delta_1, \ldots, \delta_n\}, n > 1, \)
// effect: a modification of \( \Sigma/descOf \) that resolves the conflict

case \( C \) of
(i) \textit{unamendable} conflict: at least one \( \delta_i \) is an atomic type,
print(\( \tau, p, C \), ‘please correct this inconsistency’) and exit
(ii) all \( \delta_i \) are interface names:
call Restructure\( (C) \)
if \( \exists \text{gli}(\delta_i), i = 1 \ldots n \), return
else:
(a) \textit{amendable} conflict: \( \nexists \text{ any mli}(\delta_i), i = 1 \ldots n \),
print(‘adding to \( \Sigma \): \( \eta = \delta_1 \) and \( \delta_2 \ldots \) and \( \delta_n \)’)
if suggestion is accepted, add to \( \Sigma \):
\( \eta = \delta_1 \) and \( \delta_2 \ldots \) and \( \delta_n \); return
(b) \textit{factorable} conflict: let \( D = \text{mli}(C), | D | > 1 \)
call Factor\( (\tau, p, C, D) \); return

Figure 3: The Solve procedure

Restructure // input: \( B = \{\beta_1, \ldots, \beta_s\}, s > 1 \)
// effect: a modification of \( \Sigma/descOf \) relation in \( B \)

Display all pairs \( \beta_i, \beta_j \) in \( B \), s.t.:
\( \beta_i = \sigma_1 \) and \( \ldots \) and \( \sigma_n \) and \( \{p_1 : \zeta_1, \ldots, p_m : \zeta_m\}, n, m \geq 0 \)
\( \beta_j = \sigma_1 \) and \( \ldots \) and \( \sigma_{n+k} \) and \( \{p_1 : \zeta_1, \ldots, p_{m+h} : \zeta_{m+h}\}, k, h \geq 0 \)
(subject to a sorting of \( \sigma_v \) and \( \{p_r : \zeta_r\}, v = 1 \ldots n, r = 1 \ldots m \))
print (‘\textit{directDesc}(\beta_j, \beta_i)?’)
if suggestion is accepted, update \textit{directDesc}
and if \( n > 0 \), replace \( \beta_j \) with:
\( \beta_j = \beta_i \) and \( \sigma_{n+1} \) and \( \ldots \) and \( \sigma_{n+k} \) and \( \{p_{m+1} : \zeta_{m+1}, \ldots, p_{m+h} : \zeta_{m+h}\} \)
return

Figure 4: The Restructure procedure
Factor  // input: \( \tau, p, C = typesOf(\tau, p) = \{ \delta_1, \ldots, \delta_n \}, n > 1 \),
// \( D = mli(C) = \{ \sigma_1, \ldots, \sigma_m \}, m > 1 \)
// effect: a modification of \( \Sigma/descOf \) that resolves the conflict

offer a selection between the following options:
(*) make \( \sigma_i \) the type of \( p \) in \( \tau \), for some \( i \)
i.e.: \( gli(typesOf(\tau, p)) = \sigma_i \)
(*) call Restructure(\( D \))
(*) let \( \hat{C} = \{ \delta_1, \ldots, \delta_{n+k} \} \) be the set of mui’s of \( D \)
    add to \( \Sigma \) a new interface definition
    \( \delta = \delta_1 \text{ and } \delta_2 \ldots \text{ and } \delta_{n+k} \),
    replace all occurrences of each \( \delta_j \)
    in the inheritance component of each \( \sigma_i \) by \( \delta \)
(*) exit
execute the selection
return  // unless exit was selected

Figure 5: The Factor procedure

the \( typesOf(\tau, p) \). This option was illustrated in the Example 3.2. If these options are rejected
by the designer, then a new interface is added whose position is between the set \( D \) of mli’s of \( C \) and the set of conflicts \( C \). We note that while \( D \) is a maximal set of mli’s for \( C \), the latter is
not necessarily a maximal set of minimal upper interfaces (mui’s)\(^3\) of \( D \); hence before the new
type is introduced, \( C \) is replaced by \( \hat{C} \), the set of mui’s of \( D \). The new type is defined as the
and of \( \hat{C} \), that makes it, in particular, the gli of \( C \), hence it becomes also the type of \( p \) for \( \tau \).
Further, it is made the directDesc of all the elements of \( \hat{C} \).

Finally, once the schema is in \( EF \), the Compact procedure checks if there exist interfaces,
related by \( descOf \), having identical definitions. These may be, in particular, interfaces with
identical definitions that are present in the original schema from the beginning, or they may be
interfaces that have been introduced during the inheritance process (note that the Restructure
procedure, in general, deals with interfaces with implicit definitions, hence may have not been
applicable). Now, when all interfaces have been expanded, there is a new chance to compact
them. This case is shown by the Example 4.1, presented in the next subsection.

It is easy to verify that the following proposition holds.

**Proposition 4.1.** If the algorithm Expand succeeds, then the resulting schema is well-formed.
\( \square \)

4.1. One more example

In this subsection, one example is presented consisting of a schema that contains a number of
inheritance conflicts. The algorithm is able to process it and produces a restructured hierarchy
that is free of inheritance conflicts and, furthermore, looks much more intuitive. In particular,
a new interface is introduced that, subsequently, is merged with an existing interface in the
schema, as suggested by the Compact procedure.

\(^3\)These are defined analogously to mli’s.
Compact  // input: a full explicit schema $S$
// output: $S$ (possibly modified)

for each pair $\delta, \kappa$ in $S$ s.t.:
  $i(\delta)$ and $i(\kappa)$ are identical and directDesc($\kappa, \delta$)
print ('merging ($\delta, \kappa$)')
if suggestion is accepted, update $S$
return $S$

Figure 6: The Compact procedure

Example 4.1. Consider the following schema, whose directDesc relation is graphically represented in Figure f, where in place of having a friendlyempl defined as a friend and an employee, we have:

```
person = {name : string, age : integer, friend_of : employee}
worker = {name : string, salary : integer, friend_of : friendlyempl}
friend = {name : string, age : integer, friend_of : friendlyempl}
employee = person and worker
friendlyempl = person and friend and {salary : integer, club : string}
friendlymgr = employee and friendlyempl and {project : string}
friendlysecr = employee and friendlyempl and {boss : friendlymgr}
```

In this case, the person and worker interfaces have a non-trivial conflict in correspondence to the friend_of property, arising when expanding the employee interface. The Restructure procedure does not succeed in finding a gli between friendlyempl and employee, since their definitions are incomparable, hence the Factor procedure is applied. Suppose now the designer wants to have a gli for the conflicting interfaces (he/she does not choose the first option of the Factor). Then, only the third option is applicable, since the Restructure again does not succeed in finding a gli between friendlymgr and friendlysecr. If the designer agrees, a new interface is introduced, say newlabel, and the schema is modified as follows:

```
friend    person    worker
    |        /        |
    |       /         |
    |      /          |
    |     /           |
    |    /            |
    |   /             |
    |  /              |
    | /                |
    |/                 |
    friendlyempl     employee
    |                    |
    |                    |
    |                    |
    |                    |
    |                    |
    |                    |
    |                    |
    |                    |
    |                    |
    friendlymgr  friendlysecr
```

Figure f

i.e., the directDesc relation is modified as shown in Figure g.

\[
\begin{array}{c}
\text{friend} \\
\downarrow \\
\text{friendlyempl} \\
\downarrow \\
\text{newlabel} \\
\downarrow \\
\text{friendlymgr} \\
\downarrow \\
\text{friendlysecr}
\end{array}
\]

Figure g

Once all the conflicts have been solved, the full EF schema is given by the following definitions:

\[
\begin{align*}
\text{person} &= \{ \text{name : string, age : integer, friend_of : employee} \} \\
\text{worker} &= \{ \text{name : string, salary : integer, friend_of : friendlyempl} \} \\
\text{friend} &= \{ \text{name : string, age : integer, friend_of : friendlyempl} \} \\
\text{employee} &= \{ \text{name : string, age : integer, salary : integer,} \\
& \hspace{1cm} \text{friend_of : employee} \} \\
\text{friendlyempl} &= \{ \text{name : string, age : integer, friend_of : newlabel,} \\
& \hspace{1cm} \text{salary : integer, club : string} \} \\
\text{friendlymgr} &= \{ \text{name : string, age : integer, friend_of : newlabel,} \\
& \hspace{1cm} \text{salary : integer, club : string, project : string} \} \\
\text{friendlysecr} &= \{ \text{name : string, age : integer, friend_of : newlabel,} \\
& \hspace{1cm} \text{salary : integer, club : string, boss : friendlymgr} \} \\
\text{newlabel} &= \{ \text{name : string, age : integer, friend_of : newlabel,} \\
& \hspace{1cm} \text{salary : integer, club : string} \}
\end{align*}
\]

and the directDesc relation is represented in Figure g. Now, having the explicit interfaces, we note that the definition of the newlabel interface coincides with the one of friendlyempl,
hence applying the *Compact*, a further suggestion is provided, that consists in a modification of the definitions of the schema as follows:

**Def:**

```
person = \{ name : string, age : integer, friend of : employee \}
worker = \{ name : string, salary : integer, friend of : friendlyempl \}
friend = \{ name : string, age : integer, friend of : friendlyempl \}
employee = \{ name : string, age : integer, salary : integer, 
                friend of : employee \}
friendlyempl = \{ name : string, age : integer, friend of : friendlyempl, 
                 salary : integer, club : string \}
friendlymgr = \{ name : string, age : integer, friend of : friendlyempl, 
                 salary : integer, club : string, project : string \}
friendlysecr = \{ name : string, age : integer, friend of : friendlyempl, 
                 salary : integer, club : string, boss : friendlymgr \}
```

The Figure h represents the inheritance relation after the last modification, taking into account that it must verify the Def. 2.2 of OODB schema (i.e., the pair $directDesc(friendlyempl, person)$ must be removed).

### Figure h

![Inheritance Diagram](image)

5. Conclusion and Future Work

Object-oriented design is important not just for object-oriented schema design but in the more general context of information system design. Although the object-oriented paradigm has the advantage of offering to a designer a model that can reflect reality quite naturally, the design activity itself still faces problems and difficulties. Several of these are related to multiple inheritance and the conflicts it may introduce. In this paper we have addressed one of these difficulties, namely the resolution of type information in the presence of multiple inheritance conflicts.
Our approach is distinguished by the combination of a theoretical basis and a sound heuristic. The theoretical basis includes precise definitions for OODB semantics, and the use of a branding mechanism to introduce a notion of type equivalence by name, which allows a fine distinction between types with the same or similar structures. However, rather than requiring designers to explicitly provide branding declarations, these are inferred by the system from the given type declarations. In addition to this implicit branding mechanism, we presented a system of heuristic-driven algorithms that discover inconsistencies, analyze them into different kinds, and offer feasible solutions, all the while preserving the freedom of a designer to choose his/her own way. We believe that our approach provides a sound and intuitive basis for addressing an important class of schema design scenarios. The proposed approach is being included in ProForm [26], a Knowledge-Base system for enterprise modeling, developed at IASI-CNR in Rome.

While we have discussed our approach only for the data model supported by ODMG, there seems to be no obstacles to extend it to attributes with combination of set and tuple types.

References


