Inheritance Conflicts in Object-Oriented Systems

Tok Wang LING and Pit Koon TEO

Department of Information Systems and Computer Science
National University of Singapore

Abstract. Inheritance conflicts in class hierarchies are avoidable through design that considers application semantics. An algorithm, in the form of IF-THEN statements, is presented that resolves any name conflicts in class hierarchies. Several examples are given which show that conflicts arise because of poor design. These conflicts can be resolved by redesigning the schema, renaming properties, redefining (or overriding) an overloaded property, factoring attributes to a more general class, removing redundant ISA relationships and/or explicitly selecting an inheritance path.

1 Introduction

In the object-oriented (OO) paradigm, classes related through the ISA relationship are organised into a class hierarchy. The class hierarchy provides an inheritance mechanism which allows a class to inherit properties (attributes/methods) from its superclasses, if any. In a single inheritance situation, a class A has only one direct superclass B. If both A and B define a property p, then A has a conflict situation which is resolved by adopting a precedence rule that chooses A's property. This is the generally accepted way of handling single inheritance situations. In a multiple inheritance situation, a class A has two or more direct superclasses B1, B2, .., Bn (n>1). Consider the case in which each Bi ($1 \le i \le n$) has commonly named properties p1,..,pk ($k \ge 1$). Many OO database systems (OODBMSs) treat this as a conflict situation and provide several techniques to resolve it, e.g. choosing the first in a list of superclasses[6], using type information[5], explicitly choosing the required property to inherit[4] etc.

In this paper, we show that these techniques are not satisfactory because they do not examine the reasons why conflicts arise. We propose that these conflicts are avoidable if application semantics are considered during the schema design stage. An algorithm, in the form of IF-THEN statements, is presented that resolves any name conflicts in class hierarchies. The approaches we adopt include redesigning the schema, renaming properties, redefining an overloaded property, factoring attributes to a more general class, removing redundant ISA relationships and explicitly selecting the desired property.

Section 2 provides some background information while Section 3 provides a motivating example. In Section 4, a model of inheritance is proposed. A design algorithm is given in Section 5 which resolves conflicts in ISA hierarchies. Examples are given in Section 5 to substantiate our approach. Section 6 concludes. Many of the examples are illustrated using Entity Relationship diagrams[3].

2 Background

Several interpretations of the ISA relationship exist[2]. One such interpretation imbues the class hierarchy with a set inclusion semantics: Given classes A, and B, A ISA B implies that an instance of A is also an instance of B. This interpretation is adopted in situations where the class hierarchy is used to organise and classify concepts. Such a class hierarchy has been called a classification hierarchy[9]. Classification hierarchies are typically used to model ISA relationships in database schema design. When the class hierarchy is used as a model of concept classification, the order of superclasses is insignificant in a multiple inheritance situation. For example, consider the classic University database schema in which a TEACHING_ASSISTANT class is a subclass of both EMPLOYEE and STUDENT classes. The order of STUDENT and EMPLOYEE does not, and should not affect the semantics of the subclass TEACHING_ASSISTANT.

Another interpretation treats each subclass of a class hierarchy as a specialisation of its superclass(es). Therefore, given classes A and B, A ISA B implies that A inherits all properties of B (possibly redefined). Further, A can specify some extra properties beyond what B possesses. This interpretation is typically adopted in situations where a superclass shares code with its subclass(es). Such a hierarchy has been called a specialisation hierarchy[9]. Code sharing via the ISA hierarchy is predominantly used in OO programming languages. In some of these languages, e.g. Flavors, to handle code sharing in multiple inheritance situations, an ordering is imposed on the superclasses so that a class that inherits from superclasses A and B is different from a class that inherits from B and A. Intuitively, this is a straightforward way of handling multiple inheritance in a programming language, and it has been adopted in some OODBMSs, e.g. ORION[6], to handle conflicts in class hierarchies. This approach is, however, inadequate, as will be discussed in the next section.

An important property of the ISA relationship is that it is transitive, i.e. given classes A, B and C such that A ISA B and B ISA C, we can infer that A ISA C, i.e. an instance of A is also an instance of C.

3 Motivating Example

Several resolution techniques have been proposed for OODBMSs to handle conflicts in multiple inheritance situations[4,5,6,7]. An example of such a situation is given in Figure 1, which is adapted from [6]. The class SUBMARINE needs to determine which 'SIZE' attribute to inherit from its two direct superclasses. The method used in ORION[6] is to choose the first in the list of superclasses. An order is, therefore, imposed on the superclasses of SUBMARINE so that 'SIZE' is taken from, say, WATER_VEHICLE if it is the first in the list. This approach is somewhat arbitrary and may not yield the required semantics. For instance, given that superclasses A and B of a class C have commonly named attributes p and q, then no ordering of A and B will allow C to inherit A's p and B's q. POSTGRES [7] does not allow the creation of a subclass that inherits conflicting attributes (e.g. SUBMARINE). This approach is not flexible when compared to that in O2, which allows the explicit selection of the properties to inherit. O2 emphasises the path along which the property is to be inherited from. We feel, however, that the path is not important compared to the identity of the source class which defines (or redefines) the property and from which the property is inherited. For example, if 'SIZE' is defined in VEHICLE, but not in MOTORISED_VEHICLE and WATER_VEHICLE, then SUBMARINE inherits 'SIZE' from VEHICLE directly. There are two paths to VEHICLE from SUBMARINE, but either path will lead to the same result. IRIS[5] uses type information to resolve conflicts, e.g. by choosing a more specific type to inherit. It is not clear how type specificity is determined in IRIS.

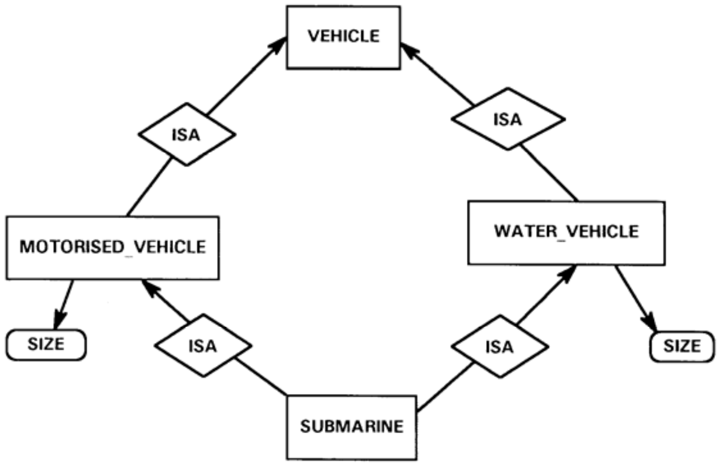

Fig. 1. Motivating Example

Many of these techniques do not examine the semantics of the properties involved in a conflict situation. The reasons why conflicts occur are not examined. Further, these techniques resolve conflicts only after the schema has been completed and implemented. In Section 5, a design methodology is proposed which resolves conflicts at the design stage. Our approach is thus proactive, as opposed to the reactive approach used in most OODBMSs. Although we recommend identifying and resolving inheritance conflicts during the design stage, there may be valid reasons for implementing a schema without removing all conflicts (e.g. user requirements). The reactive techniques of [4,5,6,7] can then be used to resolve such conflicts.

4 A Model of Inheritance

A property is *specified* in a class if it is either defined or redefined for the class. A *redefined* property overloads a similarly named property in some superclass(es) of the class. A class can inherit properties from its superclasses, if any. An inherited property is *well-defined* if it is specified in one and only one superclass, possibly indirect. A *conflict situation* exists when an inherited property is not well-defined, i.e. two or more superclasses specify the same property.

We will use ER diagrams to illustrate our notion of inheritance conflicts. Classical ER diagrams are structural and do not have the notion of methods. However, for ease of explanation, methods are depicted in our ER diagram in a similar manner as attributes. Only properties specified in an entity (class) are represented in ER diagrams.

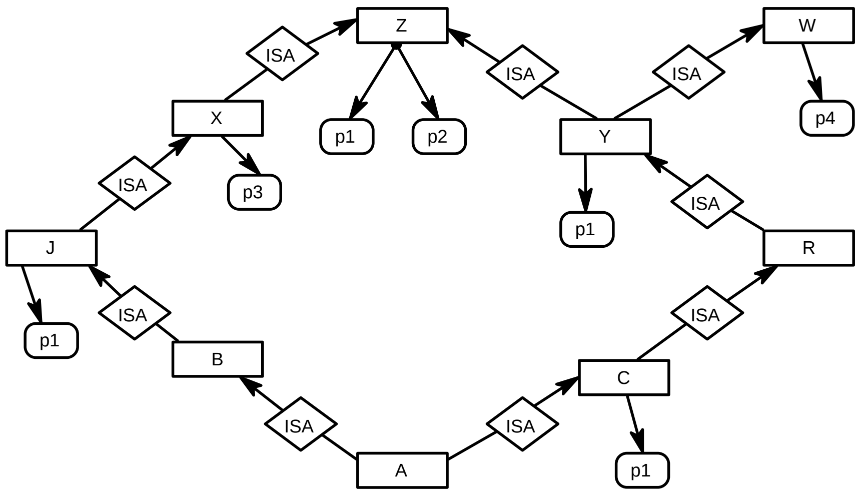

Fig. 2. An Inheritance Diagram

In Figure 2, p1 and p2 are defined in class Z, and p3 and p4 are defined in classes X and W respectively. Property p1 is redefined in classes Y, J and C and hence are explicitly represented as properties of these classes in the ER diagram. Class B inherits p1 from class J and p2 and p3 from classes Z and X respectively, while class C inherits p2 and p4 from classes Z and W respectively. Since these properties are inherited rather than specified in the classes, they are not explicitly shown on the ER diagram. Note that classes B and C have commonly named properties p1 and p2, but only p1 contributes to a conflict situation in class A. Property p1 is not well defined in A (there are two classes J and C (at least) which specify p1), but p2 is well defined (only class Z defines p2). In most OODBMSs, class A would be treated as having two conflict situations, one involving p1 and the other involving p2. Furthermore, in resolving the inheritance of p2 in class A, most systems would ask for the path of inheritance, either from B or from C. In our approach, p2 is specified in class Z and therefore the path of inheritance for this case is immaterial; either way through B or C will still converge to class Z.

In our model, properties with distinct names have different semantics. We refer to this as the *unique name assumption.* If properties with distinct names have the same semantics, we rename these properties to ensure that properties with the same semantics have the same name. However, properties with the same name need not necessarily have the same semantics. For example, in Figure 2, property p1 in class

C has a different semantics from p1 in class Y. Whenever a property p is renamed to p', it is important to rename all properties with the same semantics as p to p' in order to ensure the same consistency of meaning. This observation is important because renaming is one way to resolve conflicts in inheritance hierarchy, as we will see in the next section.

5 Avoidance of Conflicts

In this section, we introduce a design methodology which resolves conflicts in ISA hierarchies at the design stage. Our design methodology is guided by a set of IF-THEN statements, covering both single inheritance (SI) and multiple inheritance (MI) situations. Briefly, conflicts in SI situations are resolved by choosing the properties of a subclass over similarly named properties in its direct superclass. Conflicts in MI situations arise because of the following reasons. First, there may be redundant ISA relationships. By removing redundant ISA relationships, conflicts are resolved. Second, the design may be poor or erroneous. For instance, for a given subclass, the intersection of its superclasses may be empty, which suggests an erroneous design. A schema redesign will resolve the conflict. Third, given a subclass, some properties of its superclasses may have the same name and semantics. To resolve such a conflict, we either explicitly choose a superclass to inherit the property from or use a process called factoring (see Section 5.4) which moves a conflicting property to a more general class in order to resolve the conflict. Fourth, given a subclass, some properties of its superclasses may have the same name but different semantics. Renaming is recommended to resolve the conflict. The detailed algorithm is given below. Examples are given in subsequent subsections to describe the situations covered by the algorithm.

```
Given an OO schema design with ISA hierarchies,
FOR each conflict situation in the hierarchy DO
 IF it is a single-inheritance situation THEN       /* Case I : SI  (Section 5.1) */
    adopt precedence rule that prefers subclass properties; ensure semantics is understood
 IF it is a multiple-inheritance situation THEN
     /*  Check for ISA redundancy  arising from ISA transitivity property  */
   IF conflict arises because of ISA redundancy THEN
      /*  Case II : MI with ISA Redundancy (Section 5.2)  */
        resolve conflict by removing ISA redundancy
   ELSE
      BEGIN
         Let the MI conflict situation be represented by classes A, B1,..,Bn (n>1)
         where B1,..,Bn are the nearest superclasses of A that specify a property p.
         /*   Note that a superclass of some Bi may itself specify a property p.   */
         /*   Check the semantics of p in B1,..,Bn   */
         IF semantics of p is the same in B1,..,Bn THEN
         BEGIN    /* Check if the intersection of B1,..,Bn is empty */
            IF intersection of B1,..,Bn is empty THEN
            /*  Case III: MI - same semantics (Empty Subclass) (Section 5.3) */
               Design Error, since class A (which is the intersection of B1,..,Bn) is empty
            ELSE      /* Case IV: MI - same semantics (Factoring)  (Section 5.4) */
              IF there exists a more general class K which is UNION of B1,B2,..,Bn THEN
                     factor  p to class K    /* for factoring see Section 5.4 */
              ELSE
                   Resolve the conflict by either:
                   (a) creating a general class K that is the UNION of B1,B2,..,Bn and
                       factoring p to K. Add new ISA relationships Bi ISA K for i=1,..,n.
                       For each maximal superclass Ci of Bi such that K is a superset of
                       Ci, add the ISA relationship Ci ISA K and remove the redundant
                       ISA relationship Bi ISA K.
                       IF there exists a class Y such that Y is a minimal superset of K THEN
                                      insert new ISA relationship K ISA Y.
                       /* Option (a) removes data redundancy but may create some ISA
                           redundancies which will be removed by applying Case II        */
                   OR
                   (b) Explicitly choosing one superclass to inherit the property.
                        /* data redundancy exists which must be managed.   */
        END
        ELSE
          BEGIN     /* Case V: MI - properties with different semantics  (Section 5.5)*/
            Let G1,G2,..,Gm be sets of mutually exclusive classes from B1,..,Bn such that
            classes in a group share the same semantics for p. The groups G1,G2,..,Gm
            have pairwise different semantics for p. Resolve the conflict in A by:
            (a) redefining p in class A, /* not a good solution; see Section 5.5  */   or
            (b) Renaming p in Gj to, say, p_Gj for j=1,..,m to reflect their different
              semantics. To conform to the unique name assumption, each p in the
              schema that has the same semantics as p_Gj must be renamed to p_Gj.
              FOR each group Gj (j=1,..,m) with 2 or more classes having property p_Gj DO
                  /* An MI situation exists between class A and  classes in Gj;  */
                  /* p_Gj has the same semantics in the classes of Gj               */
                  Resolve the conflict in class A using the method described in cases III & IV
              ENDFOR
          END
     END
ENDFOR
```

5.1 Case I : Single Inheritance Situation

Conflict situations in single inheritance systems are relatively easy to overcome, since we can adopt the rule that the properties in a subclass take precedence over similarly named properties in its direct superclass. However, it is important to ensure that application *semantics* are well understood. Consider the ER diagram in Figure 3. From a conventional database design viewpoint, Figure 3 is erroneous because, given that MANAGER ISA EMPLOYEE, it is strictly not correct to have 'PHONE#' as a single valued attribute in EMPLOYEE and as a multivalued attribute in MANAGER. The correct semantics, from this perspective, is that the attribute 'PHONE#' in EMPLOYEE should also be multivalued. This multivalued attribute is then inherited directly by MANAGER, without any redefinition. In this case, the majority of employees will have only one telephone number as the value of the multivalued attribute 'PHONE#'. However, this semantics cannot express the fact that only a manager can have multiple 'PHONE#'. The OO approach allows us to express this fact by making use of the ability of a subclass to redefine an inherited attribute. In Figure 3, for instance, MANAGER can override the single-valued attribute 'PHONE#' from EMPLOYEE and redefine it as a multivalued attribute 'PHONE#'. This interpretation retains the semantics of the application more closely than defining 'PHONE#' as multivalued in EMPLOYEE.

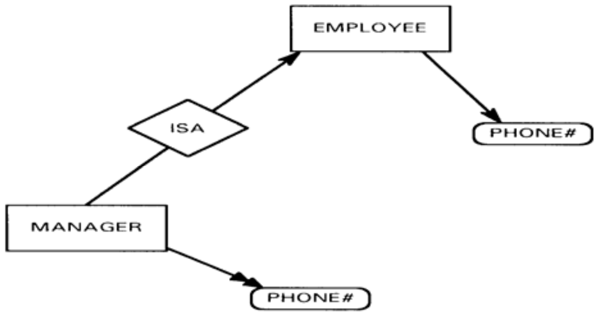

Fig. 3. 'PHONE#' is overridden in MANAGER and treated as multi-valued

5.2 Case II : Multiple Inheritance with ISA Redundancy

Redundant ISA relationships arise because of the transitive property of ISA, i.e. given A ISA B, B ISA C and A ISA C, A ISA C is redundant since it can be deduced from the other two ISA relationships. Many multiple inheritance situations can be reduced to a straightforward single inheritance situations by removing redundant ISA relationships, as illustrated in Figure 4. Figure 4 is adapted from [8] and shows an example in which the value of the color attribute for CIRCUS_ELEPHANT is supposedly ambiguous. However, note that the ISA link between ELEPHANT and CIRCUS_ELEPHANT is redundant and can be removed.

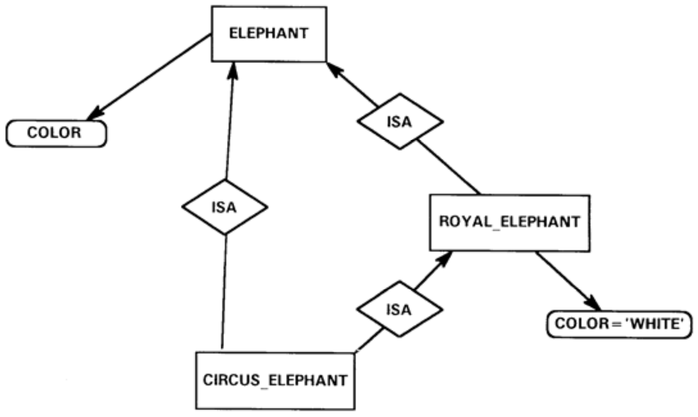

Fig. 4. Removing Redundant ISA Relationship

5.3 Case III : Multiple Inheritance - Same Semantics (Empty Subclass)

Consider classes A, B1,B2,..,Bn such that B1,B2,..,Bn are the nearest superclasses of A that specify a property p. If the semantics of p is the same in B1,B2,..,Bn and the intersection of B1,B2,..,Bn is empty, then there is a design error. This is because class A (the intersection of B1,B2,..,Bn) is empty. Figure 5 illustrates this situation.

The example schema shown in Figure 5 is adapted from [8]. In conventional database design, instances are not explicitly represented in schemas. Therefore, we assume that NIXON in Figure 5 refers to a class of Nixon-like people. In Figure 5, an instance of QUAKER is a pacifist while an instance of REPUBLICAN is not. If an instance of NIXON is both an instance of QUAKER and an instance of REPUBLICAN, then we have a potential conflict in that instance's belief. If the property PACIFIST has the same semantics in both QUAKER and REPUBLICAN, then there is clearly a design error in Figure 5, since the intersection of QUAKER and REPUBLICAN is empty, which should be rejected as a flawed design. The single-valued attribute PACIFIST can either be 'yes' or 'no' in NIXON, but not both.

Note that if the attribute PACIFIST in QUAKER has a different meaning from the attribute PACIFIST in REPUBLICAN (i.e. their semantics is different), then either choosing a specific property to inherit or changing the attribute name in either one or both superclasses will resolve the conflict. (See Section 5.5). For example, the PACIFIST attribute in REPUBLICAN can be renamed as PARTY-PACIFIST. Then an instance of NIXON will inherit two distinct attributes, viz. PACIFIST and PARTY-PACIFIST, thus removing any potential conflict.

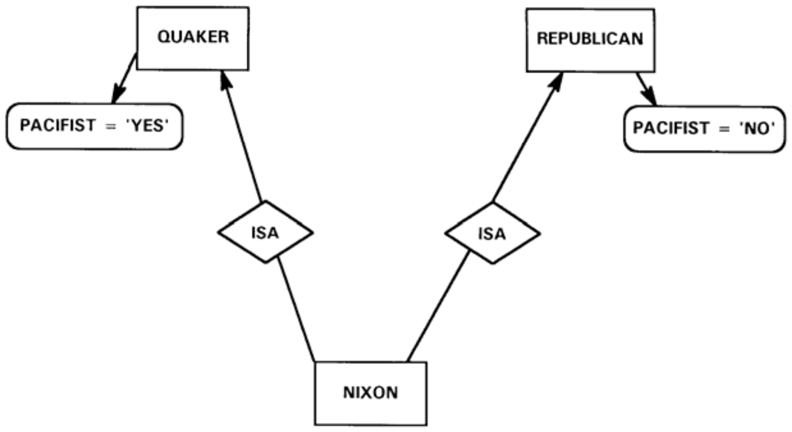

Fig. 5. 'PACIFIST' is overloaded and should be renamed to resolve conflict

5.4 Case IV : Multiple Inheritance - Same Semantics (Factoring)

Consider classes A, B1,..,Bn (n>1) such that B1,..,Bn are the nearest superclasses of A that specify a property p. In Figure 2, for instance, classes C and J are the nearest superclasses of A that specify p1. Note that it is possible for superclasses of B1,B2,..,Bn to specify a property p. For example, in Figure 2, p1 is also specified in class Z which is a superclass of C and J, and in class Y, which is a superclass of J. If p has the same semantics in B1,.., Bn, and the intersection of B1,..,Bn is not empty, the conflict in class A can be resolved by one of the following two possible cases:

(Case A) If the search for a class K in the hierarchy such that K is the union of B1,..,Bn succeeds, then perform the following steps:

(i) If K has a property p, rename the p in K to p'. Note that p' has a different semantics from the p in B1,..,Bn. For each p in the schema that has the same semantics as p', rename p to p'. This conforms to the unique name assumption (see Section 4). Then specify p in K with the same semantics as p in B1,..,Bn.

(ii) If a superclass of Bi (i=1,..,n), say Bi', lying between Bi and K, specifies a property p, then redefine the property p in Bi to explicitly inherit p from K. This prevents Bi from inheriting the property p of Bi' (whose semantics is different from p of Bi). If no superclass of Bi (i=1,..,n), lying between Bi and K specifies a property p, then remove p's definition from Bi. This ensures that Bi inherits p from K.

(iii) explicitly state that class A inherits p from K. This prevents A from inheriting p from any superclass of A lying between A and K that specifies a property p.

The execution of steps (i), (ii) and (iii) is called *factoring* p to K. Note that there can be multiple paths from A to K, but it is only important to note that A's p is taken from K.

(Case B) If no class K can be found such that K=UNION(B1,..,Bn), then consider creating a class K such that K=UNION(B1,..,Bn) and inserting K into the class hierarchy together with ISA relationships Bi ISA K, for i=1,..,n. Then factor p to K. For each *maximal* superclass Ci of Bi such that K is a superset of Ci, (i.e. there does not exist a superclass Ci' of Bi such that K is a superset of Ci' and Ci' is a superclass of Ci), we add the ISA relationship Ci ISA K and remove the redundant ISA relationship Bi ISA K. For any class Y such that Y is a *minimal* superset of K(=UNION(B1,..,Bn)), i.e. there does not exist a superset Y' of K such that Y' is a proper subset of Y, then insert the ISA relationship K ISA Y. Note that redundant ISA relationships may be created. However, these redundancies will be removed by using the method in Section 5.2.

In Figure 2, for instance, let the semantics of p1 in classes C and J be the same. Then class A has a conflict situation which can be resolved as follows:

(Case A) If Z=UNION(C,J), we (i) rename p1 in Z to p1', rename each p1 in the schema with the same semantics as p1' to p1', and specify p1 in Z with the same semantics as that in C and J, (ii) remove p1 from J and redefine p1 in class C so that p1 is explicitly inherited from Z; this prevents class C from inheriting p1 from Y, (iii) state in class A that p1 is explicitly inherited from Z; this prevents class A from inheriting p1 from class C. Note that there are two paths from A to Z, but either path will yield the same result.

(Case B) If Z is a superset of UNION(C, J), consider creating a class K such that K=UNION(C,J) and inserting it into Figure 2, together with the ISA relationships J ISA K, C ISA K and K ISA Z. Then factor p to K. For each maximal superclass S of C such that K is a superset of S, insert S ISA K and remove the redundant relationship C ISA K. For instance, assuming that K is a superset of R, Y and W

(the superclasses of C) respectively, then S = W (i.e. W is maximal) and therefore we add W ISA K and remove C ISA K. Similarly, if X is a maximal superclass of J such that K is a superset of X, we insert X ISA K and remove J ISA K.

As another example, assume in Figure 1 that 'SIZE' has the same semantics in both MOTORISED_VEHICLE and WATER_VEHICLE. It is better to factor 'SIZE' to a more general class, say VEHICLE, provided all vehicles have 'SIZE' as an attribute, i.e. VEHICLE is a union of MOTORISED_VEHICLE and WATER_VEHICLE. If VEHICLE is not a union of MOTORISED_VEHICLE and WATER_VEHICLE, then create a class K which is a union of these two classes, and K ISA VEHICLE. Three ISA relationships need to be created, viz. MOTORISED_VEHICLE ISA K, WATER_VEHICLE ISA K and K ISA VEHICLE. Then 'SIZE' can be factored to K. Moreover, the two redundant ISA relationships MOTORISED_VEHICLE ISA VEHICLE and WATER_VEHICLE ISA VEHICLE will be removed by our algorithm, as described in Case II (Section 5.2).

Sometimes, it may not be agreeable to create additional classes in order to resolve conflicts. In this case, the user may explicitly choose to inherit from one of B1,..,Bn. However, this option means that for each instance of A, the property p is redundantly stored in each of B1,..,Bn. For example, in Figure 1, the user may choose to select the 'SIZE' attribute from, say, WATER_VEHICLE. Then, for each instance of SUBMARINE, the 'SIZE' attribute is redundantly represented in MOTORISED_VEHICLE and WATER_VEHICLE. This redundancy is clearly not desirable.

5.5 Case V : Multiple Inheritance (Properties with Different Semantics)

Consider classes A, B1,B2,..,Bn such that B1,B2,..,Bn are the nearest superclasses of A that specify a property p. Let G1,G2,..,Gm be mutually exclusive groups formed from B1,B2,..,Bn such that each class in a group shares the same semantics for p. The semantics of property p is different across the groups G1,..,Gm. To resolve the conflict in class A, we propose that one of the following two options be adopted:

(Option A) the user can redefine or overload the property p in class A. Note that in some systems, e.g. O2, the user can also explicitly select a particular p to inherit from one of G1,..,Gm. We do not advocate either redefinition or explicit selection of p, because it precludes A from inheriting p with different semantics from G1,..,Gm.

(Option B) the property p can be renamed in each group Gj (for j=1,..,m) to p_Gj, since they have different semantics. Each p in the schema that has the same semantics as p_Gj must also be renamed to p_Gj. This is to conform to the unique name assumption (Section 4). For each group Gj (j=1,..,m) with two or more classes having property p_Gj, a multiple inheritance situation exists between A and the classes in Gj. Since the semantics of p_Gj is the same for the classes in Gj, the techniques in Sections 5.3 and 5.4 can be used to resolve the conflict in class A.

As an example, consider Figure 1 again. If the interpretation of 'SIZE' in MOTORISED_VEHICLE and WATER_VEHICLE is 'engine size' and 'capacity' respectively, then the semantics of 'SIZE' in these two classes is clearly different. The two approaches described above can be used to resolve this conflict: First, the user can redefine the 'SIZE' attribute to have its own semantics in SUBMARINE. It is also possible for the user to explicitly select a required interpretation of 'SIZE' for SUBMARINE, i.e. choose the superclass required. Either way, it is not possible to access both the 'engine-size' and 'capacity' of an instance of SUBMARINE. Second, the user can rename the 'SIZE' attribute in either one or both superclasses, and inherit the renamed attribute(s). This conforms to the unique attribute name assumption and allows an instance of SUBMARINE to have both interpretations of 'SIZE'.

5.6 Conflicts in Specialisation Hierarchies

It is possible to define two classes A and B such that A inherits B's code without necessarily implying that each of A's instance is also an instance of B, i.e. there is no set inclusion semantics. Consider the following two class definitions from [1] (using a C++ like notation):

```
class supplier {        Name nm;
                        Addr addr;
            public :    supplier(Name xname, Addr xaddr); /* constructor */
                        Name name();
                        Addr address(); }
class item {            Name nm;
            public :    item (Name xname); /* constructor */
                        Name name(); }
```

In [1], a class stockitem is subsequently defined as a derived class of both supplier and item.

```
class stockitem: public item, public supplier {
                        int consumption;
            public :    int qty;
                        stockitem(Name iname, int xqty, int xconsumption,
                                  Name sname, addr saddr);
                        int reorder_qty();
                        Name suppliername();
                        Name itemname();
                        Name name(); }
```

The class stockitem is specified to share code/properties with supplier and item. A conflict then arises because the method name() defined in both supplier and item has the same name. The method used in [1] is to use explicit qualification to resolve this conflict. Therefore, if it is required that the name() function of stockitem yields the supplier's name, then name is redefined as:

```
Name stockitem::name()
{ return supplier::name() ; }
```

Similar changes to the above code are required if item name is required. This example illustrates that an attempt to share in object-oriented languages may produce a conflict situation which typically is resolved using some language feature (e.g. explicit qualification as the example shows).

6 Conclusion

In this paper, we showed through several examples the reasons why conflicts can arise in multiple inheritance systems. We also discussed the conflict resolution techniques adopted by several object-oriented database systems, e.g. ORION, O2 and POSTGRES. The main fault with these techniques is that they operate at a syntactic, rather than a semantic, level.

We propose that conflicts can be avoided if more thought is given to application semantics during the design stage. Conflicts are removed by redesigning the schema, by renaming the properties in order to satisfy the unique attribute name assumption, by removing redundant ISA relationships, by factoring properties to a more general class, by overriding (or re-implementing) an overloaded property and by explicitly choosing an inheritance path in order to preserve the advantage of name overloading. An algorithm, in the form of IF-THEN statements, is given to resolve conflicts in ISA hierarchy in a systematic manner.

References

1. R. Agrawal, N. H. Gehani: ODE (Object database and environment): The language and the data model. Proc. ACM-SIGMOD Intl. Conf. on Management of Data. 1989, pp. 36-45.
2. R.J. Brachman: What IS-A is and isn't. Computer. Vol 16 No 10, Oct 1983, pp. 30-36.
3. P.P. Chen: The entity-relationship model: toward a unified view of data. ACM Transactions on Database Systems. Vol 1 No 1, 1976.
4. O. Deux et al: The story of O2. IEEE Transactions on Knowledge and Data Engineering. Vol 2 No 1, Mar 1990, pp. 91-108.
5. D. Fishman et al: IRIS: An object oriented database management system. ACM Transactions on Office Information Systems. Vol 5 No 1, Jan 87, pp. 48-69.
6. W. Kim: An introduction to object-oriented databases. MIT Press, Cambridge, Mass., 1990.
7. L. Rowe, M. Stonebraker: The Postgres data model. The Postgres Papers, Memo No. UCB/ERL M86/85. University of California, Berkeley, Jun 1987 (Revised).
8. D. Touretzky: Implicit Ordering of Defaults in Inheritance Systems. Proc. AAAI-84, Austin, Texas, 1984, pp. 322-325.
9. S. Zdonik D.Maier: Fundamentals of object-oriented database systems. Readings in Object-Oriented Database Systems. Morgan Kaufman, San Mateo, Ca., 1990, pp. 1-32.