Modeling Class Hierarchies with Contradictions

Alexander Borgida
Department of Computer Science
Rutgers University
New Brunswick, NJ 08903

Abstract

One characteristic feature of object-oriented systems and knowledge bases (semantic data models, conceptual modeling languages, AI frames) is that they offer as a basic paradigm the notion of objects grouped into classes, which are themselves organized in subclass hierarchies. Through ideas such as inheritance and bounded polymorphism, this feature supports the technique of *"abstraction by generalization"*, which has been argued to be of importance in designing Information Systems [11, 2].

We provide in this paper examples demonstrating that in some applications *over-generalization* is likely to occur: an occasional natural subclass may contradict in some way one if its superclass definitions, and thus turn out not to be a strict subtype of this superclass. A similar problem arises when an object is allowed to be a member of several classes which make incompatible predictions about its type. We argue that none of the previous approaches suggested to deal with such situations is entirely satisfactory.

A language feature is therefore presented to permit class definitions which contradict aspects of other classes, such as superclasses, in an object-based language. In essence, the approach requires contradictions among class definitions to be *explicitly* acknowledged. We define a semantics of the resulting language, which restores the condition that subclasses are both subsets and subtypes, and deals correctly with the case when an object can belong to several classes. This is done by separating the notions of "class" and "type", and it allows query compilers to detect type errors as well as eliminate some run-time checks in queries, even in the presence of "contradictory" class definitions.

1. Introduction

A major advantage of object-oriented approaches to programming is that they allow programmers to set up a *direct and natural* correspondence between program components and the concepts of the application domain. This is particularly important in domains such as databases and artificial intelligence, where the computer is called upon to essentially maintain a model of some portion of the world.

In fact, a variety of object-oriented languages, called Conceptual Modeling Languages [3], have evolved for the purpose of designing Information Systems; and Semantic Networks and Frame Languages are popular techniques for representing knowledge in Artificial Intelligence that are based on the concept of object.

The following are the important features of object-based languages that will be exploited in this paper:*

- All information is recorded through the presence (or absence) of ***objects***, which are related to each other through ***properties / attributes***.
- Similar objects are grouped together into ***classes***, which prescribe the properties applicable to their instances, as well as constraints on the possible values for properties.
- Classes are organized into ***subclass hierarchies***, where instances of a class A are also instances of its superclass B, and all constraints applicable to instances of B, also apply to instances of A.

The examples in the figure below illustrate the way in which classes may be (partially) defined.

```
class Address with
  street: String;
  city: String;
  state: {'AL,...,'WV};

class Person with
  name: String;
  age:  1..120;
  home: Address;

class Employee is-a Person with
  age : 16 .. 65;
  supervisor : Employee;
  office : Address;
```

Here, Employee is defined as a subclass of Person**. The technique of *inheritance* makes it unnecessary to restate the constraints on the attributes of Person for Employee, allowing the designer to concentrate on the additional attributes, such as ***supervisor***, etc. It is important to observe that a subclass may in addition refine a previous condition stated on a more general class: employee ***age***s are required to lie in a more restricted range than person ***age***s.

*Note that this paper does not deal with the orthogonal issues of encapsulation and message passing, which are often also associated with the object-oriented paradigm in programming languages but are largely absent in semantic data modeling.

**In general, a class is allowed to have more than one superclass, so that the subclass hierarchy is not necessarily a tree.

In [4], it was argued that it is in general impossible or undesirable to anticipate all possible states of the world during schema design, and therefore it is necessary to allow classes to contain as instances *exceptional individuals*: ones who do not satisfy all the constraints stated on the class definition. The proposed solution essentially provided for the special treatment of these individuals through *run-time* exception handling, and, for efficiency, relied on the *rarity* of exceptional occurrences.

In some situations, entire collections of objects can be anticipated to be "exceptional": temporary employees get lump sum payments, and do not have (monthly) salaries; executives, though employees in other ways, are supervised by members of the Board of Directors, who are not employees themselves. The case of flying birds, with a subclass of penguins, which do not fly, is probably the best known example of this in Artificial Intelligence. In such cases the cost of the mechanism suggested in [4] may seem too high.

The purpose of this paper is therefore to consider in depth the motivation for allowing subclasses that can completely redefine some attribute, as opposed to refining it, and to investigate solutions to this issue.

In order to demonstrate the utility of permitting exceptional or partially contradictory subclass definitions, we will show that the arguments in their favor rest on the same principles as the arguments supporting the use of the object-oriented approach itself. After some motivating examples, and an analysis of the problems they pose, we compile a list of desireable properties for a mechanism supporting non-strict inheritance. In Section 5 we present a simple concept -- excusing contradictions -- which we claim has the desired properties, and investigate its semantics and extensions.

2. The role of classes.

In an Information System, the class definitions provide *end-users* with a description of the "domain of discourse" of the system: they delimit the area about which the system is "knowledgeable", and define the vocabulary which one has to use to access this information. In addition, the constraints specified in class definitions help detect ubiquitous data entry errors, thus preserving the quality of the data, which is obviously in the interest of the users.

The information in the schema also allows compilers and physical database designers to set up efficient data storage and access mechanisms.

Finally, the object-based approach provides significant advantages for the *designers* of the schema and the *implementors* of application programs running over the database. It is these issues which interest us most, and for this reason we will categorize them below under the various functions that classes play.

a) The class as type definition

One can simply view classes as type definitions. Their role is then to label the domains and ranges of operations in order to detect ill-defined manipulations of the data. For example, the previous definitions indicate that salaries, unlike names, are numbers, and can therefore be used in arithmetic operations in procedures (e.g., database transactions and queries). Attributes can be viewed as functions, in which case classes viewed as types indicate their domains and ranges too: ***supervisor*** is not applicable to arbitrary persons, only to employees. For this purpose, we might use the type system introduced by Luca Cardelli [7]: it views classes as record types, allows nested record definitions, and defines an appropriate notion of "subtype", which matches the notion of subclass.

The utility of type systems and compile-time type checking for *detecting errors* in procedures is well known: For example, if we have a type checking scheme, it is possible to flag an attempt to evaluate the ***supervisor*** of an arbitrary person, who is not deducible to be an employee.

b) The class as type identifier

The advantages of strong typing (e.g., knowing what attributes are applicable to an object) accrue without associating identifiers with classes. For example, one could explicitly type the arguments of a procedure with the properties which we expect them to have. Or, we could do away with the Address class by expanding its definition "in-line":

```
class Person with
  home:  [street: String;
          city: String;    ... ]
  office: [street: String;
           city: String;    ... ]
```

However, classes in fact represent correlated attribute structures, which are referred to frequently, and which are therefore assigned unique identifiers for ease of reference. Naming classes provides: i) a way of *abbreviating* definitions; ii) a way of *localizing* definitions, which makes changes easier since we do not have to find all occurrences of the same type in the specification; iii) the ability to *specify recursive definitions*, as in the case of Employees, whose ***supervisor***s are also Employees.

As an aside, note that the ability to define types without necessarily providing identifiers for them can be exploited to allow for variations while avoiding the clutter of superfluous names. Thus, if the type of office addresses was needed in only one place, we could avoid introducing an identifier for it, and use instead the following essentially implicit definition:

```
office: Address [room# : 1..9999 ;
                 companyName : String];
```

c) The class as a set of objects of a certain type

Usually an *extent* is associated with a class, representing those objects which are instances of the class at some particular time. These objects are explicitly added or removed from the extent of this class by special operators *"create/add"* and *"remove/destroy"*. The extent records which entities exist and which do not; it also allows one to

locate entities based on some description, and to perform operations like counting entities. *Sets of objects* are the obvious conceptual data structuring technique for this purpose, since one can *quantify* or *iterate* over them, as well as test membership in them. Extents also allow the specification of definitional classes: "Employees satisfying some predicate P".

d) The class as organizer of arbitrary constraints on relationships between objects.
In addition to type constraints, there are other assertions which one would like to state as part of a logical theory of the application domain: e.g., Employees earn less than their supervisors. Such assertions can often be attached to one (or a few) classes. In this case, as well as in the case of type constraints, classes play an important role in organizing the constraints and information, and this organization is useful both during the design and the evolution of the software system.

e) Classes as objects
It is often convenient to view classes as objects themselves, so that they can be organized into meta-classes, and be assigned attributes of their own. For example, various subclasses such as Secretary, Professor, etc. might all be made instances (not subclasses!) of the meta-class Employee_Class, and each might have associated properties such as ***avgSalary*** (a property whose value might be obtained by summarizing over the extent of the class) and ***avgSalaryLimit*** (which records some policy constraint of the organization). Note that such properties are clearly not attributes of individual employees.

3. Subclass hierarchies and inheritance.

Consider an Information System for use in a hospital, which keeps track of its employees, doctors, patients, etc. The definitions presented earlier could be extended by those in the figure below.

```
class Hospital with
  location : Address;
  accreditation: {'Local, 'State, 'Federal}

class Physician is a Person with
  affiliatedWith: Hospital; ...

class Patient is a Person with
  treatedBy: Physician;
  treatedAt: Hospital; ...

class Cancer_Patient is a Patient with
  treatedBy: Oncologist;
  chemoTherapy : Drug;  ...
```

There are, of course, numerous other subclasses of patients (grouped by disease), doctors (grouped by speciality), etc.

Hierarchies of classes are evidence of a form of abstraction at work: *generalization* suggests that one abstract away the detailed differences of several class descriptions, and present the commonalities factored out as a more general superclass [11]. The resulting taxonomy of classes plays at least one important software engineering role during the development of a knowledge base: by considering class definitions starting from the top of the hierarchy, we are making use of an abstraction principle that allows details to be introduced gradually and systematically. Note that attributes and constraints stated on the superclass need not be restated for the subclasses: they are assumed to hold by **inheritance**. Thus patients and doctors also have names, addresses, etc. which are inherited from Person. During specialization, the range of an attribute (e.g., ***treatedBy***) can be modified for a subclass (e.g., Cancer_Patient), as long as the new range (Oncologist) is a proper subclass of the original range (Physician). }

The additional advantages of class hierarchies can be reviewed as before:

a) Subclasses and types
If classes are used to type the arguments of procedures which manipulate the data, then the concept of supertype is useful since it allows a form of **polymorphisms**: a procedure whose argument has type Person, will work when given an argument which has as type any subclass of Person, thereby avoiding the need for defining separate versions of the same procedure. (See [8] for a more elaborate argument on this point.)

b) Subclasses and type identifiers.
The chief advantages of inheritance appear, as before, during the evolution of the software: i) class descriptions are abbreviated, hence more readable; ii) there is less chance for inconsistencies to creep in, because the same fact is no longer repeated in several places; iii) during system maintenance, changes in a class at a higher level are known to be propagated uniformly to all subclasses; thus errors of omission are avoided by localizing changes.

Again, we can avoid cluttering the conceptual schema by introducing subtypes during specialization without requiring names to be associated with them, if they are to be used only once: cancer patients are treated by physicians certified by a special board:

```
treatedBy: Physician [certifiedBy: {'ABO}]
```

c) **Subclasses and extents.**
The class hierarchy expresses subset constraints on the extents involved: the set of all Physicians is a subset of the set of Persons. An important advantage of this constraint on the subclass relation is that it allows the extents of classes to be manipulated in a much more concise way: if an object is added to the extent of Physician, it is automatically added to the extents of all its superclasses, including Person, etc. If the extent of classes was replaced by sets (as argued in [6]), then one would need to write for every class separate procedures for adding or removing objects from its extent in order to ensure that the appropriate subset relationships would be maintained; these procedures could become sources of error as the class hierarchy evolves.

d) Subclasses and the organization of constraints. Since subclasses are supposed to describe subsets of elements of superclasses, the definition of a subclass must be consistent with that of its superclasses. Thus the age restrictions of Employees must imply the age restrictions of Persons. Both editing tools and compilers can use this as a consistency check for detecting errors in class definitions; the effect is similar to checking the consistency of procedural parameter specifications with the actual procedure arguments.

Note that many of the above advantages are built on the assumption that subclass specifications do not contradict those of superclasses. Note also that all the advantages of class hierarchies and inheritance mentioned above are strictly ones of convenience and software engineering: these concepts can be eliminated from the software without changing in any way what it can accomplish. These same criteria will play a significant role in the arguments below concerning the exact nature of class hierarchies.

4. Non-strict class hierarchies.

4.1. Problems with strict inheritance.

In understanding and describing any complex situation, it is often natural and convenient to ignore at first rare or exceptional cases, concentrating on the usual or normal situations. I have called this the *normal-case first* abstraction [4], and a chief role of exception handling facilities in programming languages is to support this.

Unfortunately, in some cases this abstraction applies naturally together with the generalization abstraction, resulting in ***over-generalizations***. A paradigmatic example of this is the commonly held beliefs that "birds fly", at the same time as "penguins and ostriches are birds" yet "penguins are flightless" and "ostriches are flightless". We shall illustrate a number of such problems arising in the domain of the hypothetical knowledge base for a hospital.

The simplest and most common form of contradiction arises when the range of an attribute for a specialized class is not necessarily a subclass of the corresponding range for the superclass. For example, one may want to define Alcoholics to be patients who, among others, are treated by psychologists:

```
class Alcoholic is a Patient with
  treatedBy: Psychologist
  ...
```

But psychologists usually are not physicians, and therefore do not have the same applicable attributes and constraints. Therefore the above is not a proper specialization of Patient. Note that the intended interpretation of this definition is not that alcoholics are treated by persons who are both physicians and psychologist (in which case there would be no contradiction) -- they need only be psychologists.

The same problem arises when an attribute becomes inapplicable to all the instances of some subclass: for example, although we may want to record a ***ward*** for Patients, this attribute would be inapplicable to Ambulatory_Patients. Although one could consider this as specializing the range of ***ward*** to the empty set, it is I believe more appropriate to state that this attribute is incorrectly applied to such patients. More generally, if for example tubercular patients were required to be treated in a specific foreign country like Switzerland, then they would be treated at hospitals which are not accredited, and which have a different kind of address:

```
class Tubercular_Patient is a Patient with
  treatedAt: Hospital
     [accreditation : None
      location: Address
                [state : None;
                 country:{'Switzerland}]]
```

Such addresses are not a proper subtype of the class Address, hence these hospitals are not a subtype of Hospital; therefore Tubercular_Patient is not a proper specialization of Patient.

Problems also arise in cases when an object may belong to several incomparable classes. For example, suppose that patients suffering from renal failure have high blood pressure **[bloodPressure : High_BP]**. On the other hand patients hemorrhaging have low blood pressure due to loss of blood: **[bloodPressure : Low_BP]**. If High_BP and Low_BP do not overlap, then we cannot have a particular object being an instance of both Renal_Failure_Patient and Hemorrhaging_Patient, although it is entirely possible that a person with renal failure would bleed. Moreover, it is part of conventional medical wisdom that such a patient would have low blood pressure, and we need ways to express such a preference in the schema.***

A variant of this problem has its origin in Artificial Intelligence: Suppose Person, Quaker and Republican are defined as follows:

```
class Person with
   opinion : {'Hawk, 'Dove, 'Ostrich};

class Quaker is a Person with
   opinion : { 'Dove } ;

class Republican is a Person with
   opinion : { 'Hawk } ;
```

Consider now the case of dick, who is both a Quaker and a Republican. Clearly, as defined above, he cannot hold any opinion without contradicting some constraint. In reality, we would probably expect dick to be either a "hawk" or a "dove", but not an "ostrich".

***If such overlaps in class membership are relatively rare, then again there is no reason to deal with them in the schema of the database: one can fall back on the handling of exceptional instances described in [4]. The point made here becomes relevant if the extent of each of the classes and their intersection has a significantly large number of instances.

4.2. Alternatives to non-strict inheritance

4.2.1. Strict inheritance with reconciliation

The most obvious solution is to generalize the portion of superclass description which is being contradicted: Patient0 could be treated by Health_Professionals, which itself would have as subclasses Physicians and Psychologists. This would allow Alcoholics, defined as before, to be a proper specialization of Patient0. Most other kinds of patients would however be treated only by physicians, so one would have to laboriously specialize the ***treatedBy*** attribute for Cardiac, Cancer, etc. patients to specify in each case Physician as a range. This essentially negates one of the significant advantages of inheritance: the factoring out of commonalities which need not be repeated.

4.2.2. Strict inheritance with intermediate classes

To recapture the advantages of inheritance, one could introduce intermediate classes whose only role is to act as anchors for inheritance. So for example, Patient_Treated_By_Physician could be specified as a subclass of Patient0, and then specialized to obtain Cardiac, Cancer, etc. patients. The first disadvantage of this solution is that it introduces definitions of dubious utility, thereby complicating the program and making it harder for users to find the useful classes they are seeking. Also, exactly such classes are likely to have uninteresting extents.

The second disadvantage is of a combinatorial nature: Suppose some class C has two attributes p and q

```
class C with   p:D ;  q:E ;
```

which need to be generalized in order to avoid contradictions with some exceptional subclasses. This would result in a new class C0, but one would need to define three specializations of it: one in which p is again restricted to D, one in which q is restricted to E, and one in which both restrictions apply. Then every time a new subclass C1 of C was added, the designer would have to decide which of the classes $C0, C0_1, \ldots, C0_3$ should actually be specialized. The problem of course becomes worse as the number of such attributes increases.

4.2.3. Dissociating classes and types

The advantages of inheritance could be retained by allowing class definitions to be derived from others in a process other than specialization, as suggested among others in [7]. Alcoholic could thus be obtained from Patient by "dropping" the original definition of ***treatedBy*** and "adding" the new one. Unfortunately, this has two important negative effects. First, polymorphism is defeated in such cases; e.g., procedures applicable to Patients cannot be applied to Alcoholics, because the latter is not a subtype of the former. Second, the extent of such a derived class is not a subset of the original class; thus quantifying over all Patients will not include Alcoholics. Both of these are extremely counter-intuitive, especially in domains where classes correspond to concepts called "natural kinds" by philosophers -- concepts which appear to have no definitions but which are part of our human experience, and for which we share certain common and frequent intuitions.

4.2.4. Default inheritance

A popular approach in Artificial Intelligence is to adopt the convention that the "closest" constraint in the hierarchy overrides all others, including ones that are contradicted. For example, if the taxonomy is a tree, then one can say that the assertion on the nearest ancestor in the taxonomy holds, and hence the inherited property can be computed efficiently by searching up the subclass tree.

This approach is a variant on the previous device of deriving textually class definitions from earlier definitions, with the advantage of being even less verbose: one does not have to explicitly "drop" the earlier specifications. Unfortunately, "default" approaches face several problems as part of a definitional mechanism. The first problem, noted already elsewhere [5], is that the search-based definition is no longer well-defined once the classes are organized in a full partial order (as opposed to a tree), or if redundant links are allowed: if class A has two ancestors, B and C, both of these could specify constraints on A by inheritance, and it is not specified which one should be chosen.

A more significant problem, from our point of view, is the fact that it is no longer possible to detect inconsistent definitions because the system cannot distinguish erroneous definitions from defaults: Whenever the definition of some class contradicts some aspect of its superclass, the contradiction could be intentional or accidental, and it is impossible to tell which is the case without further user intervention. The situation is analogous to that arising in programming languages which do not have typed variables: if x originally held a string, and y is integer valued, then is ***x:=y;*** a typo (having forgotten the quotes around y) or intentional? For the same reason, when adding a class definition somewhere in the middle of the IS-A hierarchy or when modifying a definition, it is impossible to be sure that the modified constraints will be inherited by all subconcepts, or whether some intervening concept will accidentally block the inheritance. In fact, in all languages which have "cancellable inheritance", one can find out if some property of a class is universally true only by checking all of its subclasses.**** Significantly, these problem arise especially when a large system is being modified by someone else than the original designers.

Along different lines, default inheritance is usually defined only for subclass hierarchies so that it is impossible for the definition of a class to supplant some constraint on one of its attributes (e.g., the ***treatedBy*** attribute of Tubercular_Patients) without defining a special subclass of these attribute values (Swiss_Hospital and Swiss_Address) for which the constraint in question can be cancelled. This leads once again to the proliferation of unnecessary classes.

****Brachman [5] has argued this case more extensively and cogently.

Finally, there appear to be considerable difficulties in providing clear and simple semantics for default inheritance.

5. A proposal

To summarize our previous arguments, any mechanism for dealing with non-strict specialization should have the following properties:

- inheritance: retain the abbreviatory advantages of inheritance, and the advantages it provides in localizing modifications;
- *minimality*: no extraneous classes should be introduced for purely technical reasons;
- veracity: allow the redesigner to decide when exactly a constraint will hold, without having to search blindly for contradictions;
- verifiability: the language compiler or environment should be able to alert the programmer about cases of inconsistent specification;
- locality: allow incremental changes to be made locally, without having to modify earlier definitions;
- semantics: have a simple, clear semantics, including the case of non-tree hierarchies;
- extent inclusion: the extent of an exceptional subclass should continue to be a subset of its superclass' extent;
- subtyping: subclasses should continue to be subtypes of their superclasses for the purposes of polymorphism.

5.1. A simple solution.

We believe that the difficulties of "default inheritance" are due to the fact that contradictions are not explicitly acknowledged. Suppose that we amended our language to require that if some portion of a class definition contradicts another, then an explicit excuse must be made.

For example, the proper definition of the Alcoholic subclass of Patients would include

```
treatedBy: Psychologist
    excuses treatedBy on Patient ;
```

We exploit here the fact that all parts of a class definition in an object-oriented language can be identified by a pair consisting of the name of the class and that of a property. The general intent above is to say that the constraint identified by the pair ***(Patient, treatedBy)*** for some object self in Patient, is excused from holding in the case when self is also an instance of Alcoholic.

The mechanism can be used to deal with the case of objects belonging to multiple classes, where we wish to specify a policy for dealing with contradictions. For example, to indicate that low blood pressure on bleeding patients over-rides the high blood pressure symptom of patients in renal failure, we can write the class definitions as follow:

```
class Renal_Failure_Patient with
  bloodPressure : High_BP;

class Hemorrhaging_Patient  with
  bloodPressure:Low_BP
        excuses bloodPressure
             on Renal_Failure_Patient;
```

In the example involving Quakers and Republicans, we do not wish to favor either opinion, so we write

```
class Quaker is a Person with
  opinion:{'Dove} excuses opinion
                        on Republican;

class Republican is a Person with
  opinion:{'Hawk} excuses opinion
                        on Quaker;
```

It is apparent that excuses provide a way of explicitly adjudicating between contradictory constraints. Therefore the revised rule for specialization is that if a subclass specifies a new range for an existing attribute, then this range must itself be a specialization of the inherited range(s), or it must excuse the definition(s) of the constraint(s) being contradicted. Also, if an object is an instance of several classes, then for each class C and property p specified on C, the object must either obey the constraints stated for p on C or it must be an instance of some other class which excuses this constraint.

5.2. A semantics for class definitions with excuses.

Consider first the case of strict hierarchies. As stated earlier, it is our intention to associate with each class the set of all its instances. Hence a specification of the form

```
class B with  p : R ;
```

requires that

IF x is in B THEN x.p must belong to R

Also, when class C is a subclass of class B, then each instance of C is also an instance of B. If attribute p is then redefined in C to have range S, the condition of x.p belonging to S for all instances of C is consistent with the definition of B as long as S is a subclass of R -- i.e., we have proper specialization.

The issue is how to extend this definition in the presence of excuses. In particular, we need to decide what constraint must hold exactly of every object which is an instance of two classes which make contradictory predictions about its properties. Consider therefore the abstract declarations

```
class B with  p : R ;

class E with  p : S excuses p on B ;
```

and concentrate on the conditions that must apply to attribute p for instances of B.

In case E is also declared to be **is-a** B, we wish to maintain the condition that the extent of E is a subset of B. This leads us to first attempt a semantics which simply

broadens the allowed range of p for instances of the classes being contradicted:

IF x ∈ B THEN x.p ∈ R or x.p ∈ S

This definition is inadequate because in the case of Patients/Alcoholics it becomes

```
IF  x ∈ Patient
THEN  x.treatedBy ∈ Physician   OR
      x.treatedBy ∈ Psychologist
```

which permits even non-alcoholic patients to be treated by psychologists.

One can try to be more careful by allowing deviations from the norm only when the object also belongs to an excusing class:

IF x ∈ B THEN x.p ∈ R or x ∈ E

This definition suffers from the more subtle problem that in the case of Quakers and Republicans we get

```
If x ∈ Quaker THEN x.opinion ∈ {'Dove}
        OR   x ∈ Republican

IF x ∈ Republican THEN x.opinion ∈ {'Hawk}
        OR    x ∈ Quaker
```

But then some instance, dagwood, of both Quaker and Republican would be allowed to have even opinion 'Ostrich, because neither assertion would place a condition on his opinion!

An even more stringent condition requires the excusing condition to hold exactly when an object belongs in an exceptional class:

IF x ∈ B THEN (x ∉ E AND x.p ∈ R) OR (x ∈ E AND x.p ∈ S)

This is overly restrictive in the case of Quakers and Republicans because each class points a finger at the other, insisting that the other's condition must hold. This leads us to the (correct) definition:

IF x ∈ B THEN x.p ∈ R OR (x∈ In E AND x.p ∈ S)

To paraphrase: each instance of a class must obey each attribute definition appearing on the class (or inherited) unless the instance also belongs to some class which explicitly excuses the condition in question, in which case either the original condition or the *excusing* attribute specification must hold. As we have seen above, it is not sufficient to be a member of an exceptional (sub)class for the condition to be waived.

5.3. Inheritance of excuses

On the basis of formal semantics given above, one can answer all sorts of questions concerning the behaviour of excuses.

First, as shown by the Quaker example, any specification on a class can contradict (and excuse) a constraint on any other class, including superclasses that are not immediate parents, and classes that are not IS-A related to it.

Consider now the "inheritance" of excuses. If one defines a subclass, say of Tubercular_Patients, and the exceptional attribute ***treatedBy*** is not redefined, then the excuse is inherited in the sense that nothing more need be said about this exceptional attribute. Suppose then that one defines a subclass like

```
class SpecialAlc is a Alcoholic with
    treatedBy : FOO
```

If FOO is a subclass of Psychologists, again no further excuse is necessary: the contradiction with the definition on Patient is already excused by its membership in Alcoholic class. On the other hand, nothing wrong will happen if an excuse is added -- it will simply be redundant. If FOO is *not* a subclass of Psychologist, then ***treatedBy*** needs to be excused on Alcoholic; and if FOO is not even a subclass of Physicians, then ***treatedBy*** needs to be excused on Patient as well.

The behaviour of excuses in the presence of multiple inheritance parallels the theory of individuals belonging to several classes: when a class has more than one parent, its instances must obey the constraints stated on *all* the parents, unless the class explicitly excuses some/all of the inherited constraints, *or* the ancestor classes excuse one another, as illustrated in the examples involving attributes ***bloodPressure*** and ***opinion***.

The above examples indicate that the proposed approach does not utilize in any form the topology of the inheritance hierarchy, as was the case for default inheritance for example. Further discussion of the benefits of the **excuse** mecahnism is left to the Summary section, and we content oursleves with pointing out again that all of the above determinations follow entirely from the proposed semantics.

5.4. Types for classes with excuses.

The preceding semantics for our notation emphasizes the distinction between the role of a class in Software Engineering and its role as type specifier: the definition of the class Patient does not provide a complete type for its elements until all excuses to constraints stated on Patient are also considered.

To make clear the importance of having a *type theory* corresponding to class definitions, consider a query that iterates p over the instances of Patient, and has a condition involving the expression ***p.treatedAt.location.city***. This expression will not cause any type errors, given the earlier definitions of various classes. But if it was changed to ***p.treatedAt.location.state***, then the query is no longer safe if the exceptional Tubercular_Patient subclass has been defined, because some patients are at hospitals whose address does not have a ***state*** field! On the other hand, if the evaluation of this expression was in a context guarded by a conditional test such as ***(p is not in Tubercular_Patient)***, then again type safety is restored.

A challenge for designers and implementors of languages for Information System programming is then to

design a type theory and a type inference/checking algorithm which allows the above judgements about the presence or absence of errors to be made in an automatic and efficient way. The advantages of such a theory are twofold:

- It allows the compiler to warn the user that the query/program may result in a run-time failure for certain database states.
- If "type-unsafe" queries are allowed to run, the compiler can avoid the introduction of run-time safety tests in those cases where it has determined that *no* type error can occur, and thereby considerably increase the efficiency of the code generated.

Although the full details of the type theory (type inference system, soundness and completeness, and checking algorithm) cannot be presented here, we give some intuitions about it and the kinds of reasoning supported.

In the case of class hierarchies without exceptions, we follow [7] in devising a type system that includes as types primitives, the class identifiers defined, and type constructor [p : T], representing records with attribute p of type T.

Over this set of types one can then define the **subtype relation** <, which is interpreted as subset in the semantics of types. We thus arrive at a simple theory of types. In such a context, the definition

```
class Patient is a Person with
  treatedAt: Hospital;
```

is translated into the following formulas of the theory

```
Patient  <  Person
Patient <    [ treatedAt   :  Hospital ]
```

This theory is then extended to allow expressions in some query language to be assigned types. So for example, one can conclude that if ***x*** is of type Patient then ***x.treatedAt*** is of type Hospital.

To deal with exceptions in the class hierarchy, first extend the type system to allow "conditional types":

- if T0,T1,... are types and E1,... are class identifiers then the following is also a type

$$[p : T0 + T1/E1 + \ldots]$$

The denotation of such a type is the set of objects z such that z.p belongs to T0, or z belongs to E1 and z.p belongs to T1, or ... So for example,

[salary : Integer + None / Temporary_Employee]

is a type.

As expected, one uses such types in describing excuses to constraints on properties. Thus the definitions of Patient and its subclasses presented earlier yield the subtype assertion

Patient < [treatedBy: Physician + Psychologist/Alcoholic]

Of course, in the proposed type theory

[treatedBy : Cardiologist] < [treatedBy : Physician]

will be deducible from Cardiologist < Physician, while

[treatedBy : Physician] <
[treatedBy: Physician + Psychologist/Alcoholic]

will be a theorem.

During program analysis one then accumulates information about the membership or non-membership of the value of some expression in classes and uses this to deduce further information. For example, if at some point we know that x is in the class Patient, then we can conclude that ***x.treatedBy*** is in any class which is a superclass of *both* Physician and Psychologist; when analyzing the expression

```
when x is in Alcoholic
  then ...     (*)
  else ...     (**)
```

in the (*) branch we should know that the type of ***x.treatedBy*** is Psychologist, while in (**) it is Physician. Conversely, knowing that ***y.treatedBy*** is not in Physician, and y is not in Alcoholic, should allow the deduction that y is not in Patient at all.

The details of the type reasoning system which performs the above deductions soundly, completely and efficiently (order of low polynomial) will be presented in a forthcoming paper.

5.5. A word on storage.

We consider briefly the effect of exceptional subclasses on the data structures used to achieve efficient storage and retrieval for large collections of class instances [1, 9].

A standard technique for storing information about objects is to create logical records which have as fields the attributes defined on some class -- the so called "semantic grouping" of Daplex [9]. If exceptional classes are allowed, then the record fields must be allowed to store any of the values encountered for subclasses -- i.e., we need to look at the type constraint for each attribute as described in the previous section.

For those cases where the normal and exceptional values are both subtypes of the class ANY_ENTITY (as opposed to being INTEGERs, etc.), there is no problem with the existing techniques because entities are assigned internal identifiers (surrogates) by the system and these do not normally vary structurally from class to class. Thus nothing new needs to be done as far as storage in dealing with cases like the ***treatedBy*** attribute.

Difficulties arise only when some attribute may be filled by values from incompatible types (INTEGER vs. ENTITY vs. String vs. various enumerations vs. various tuple structures), where we run the problem of having different values with indistinguishable bit-string representations, or widely differing storage requirements. In such cases, the obvious solution is to perform some form of "horizontal partitioning": store objects in the exceptional subclass in a logical file with a distinct record format. Such partitioning is in fact quite easy to express in the language of fragments

provided for the distributed implementation of Daplex [10]. This does imply that it is no longer possible to associate with every attribute a single table where all its values are stored. However, once again the type deduction algorithm can then help reduce the run-time search for the file where some particular object's attribute value is located.

5.6. More complex excuses

We wish to extend here the previous technique and semantic analysis to the case of more complex exceptions involving embedded attribute definitions, such as the ones about the accreditation and location of hospitals treating tubercular patients.*****

Once again, we desire to acknowledge syntactically the conflict, and indicate which condition is to hold. In the case of tubercular patients and their hospital's accreditation, one way to accomplish this would be to add excuses pointing to the most specific conditions being contradicted in each case:

```
class Tubercular_Patient is a Patient with
  treatedAt: Hospital
    [accreditation : None
        excuses accreditation on Hospital ;
     location: Address
                [state : None
                   excuses state on Address;
                 country : {'Switzerland};
                ];
    ];
```

Embedded specifications set up "virtual classes"; the above would result in the definition of: i) an (exceptional) subclass of Address, with no ***state*** attribute, but an additional ***country*** attribute:

```
class A1 is a Address with
    state : None  excuses state on Address;
    country : {'Switzerland};
```

ii) an (exceptional) subclass of Hospital, which does not have ***accreditation***, and has a refined ***address*** attribute:

```
class H1 is a Hospital with
  accreditation : None
      excuses accreditation on Hospital ;
  location : A1;
```

With these implicit classes, the definition of Tubercular_Patient no longer has unresolved contradictions: ***treatedAt*** is properly specialized to have range H1.

The semantics of these nested excuses is complicated by the fact that virtual classes such as H1 and A1 are not *explicitly* manipulated, and hence we need an alternate way of detecting when an object belongs to their extent. The solution is to view the extent of H1 to be exactly those objects which are the values of ***treatedAt*** attributes for some Tubercular_Patient. Similarly, A1 has as extent the range of values for ***location*** attributes of instances of H1. In this sense, the extent of such virtual classes is implicitly manipulated when explicit changes to normal classes are made.

*****To remind the reader: such patients are treated at Swiss hospitals, which do not have an ***accreditation*** and whose locations have unusual fields, such as ***country***, as well as missing fields such as ***state***.

6. Summary.

We started by arguing that the same reasons which lead us to use in some programming context classes of objects organized into subclass hierarchies, would also lead us to desire under natural circumstances the ability to define subclasses which contradict some aspects of their superclasses -- i.e., are exceptional in some respect.

We then reviewed the difficulties encountered by current approaches to this problem, and consequently proposed the syntax of explicit *excuses* as a solution to these problems. Let us reconsider here this solution in light of the various desiderata we had presented in the earlier section:

As the examples show, we are able to deal with all the different cases of non-strict class hierarchies without losing the benefits of inheritance, and without having to introduce superfluous classes for strictly technical reasons. The only cost is that of explicit excuses.

Exactly because of explicit excuses, it is possible and in fact easy to detect mistakes in subclass definitions: a redefinition of an attribute which is not a specialization is an error without an accompanying excuse. Similarly, a modification to some class definition is propagated to all its subclasses; this may result in unexcused contradictions being found by the compiler/environment, which the designer must address explicitly.

By having the exceptional situation excuse the general condition, rather than the opposite direction, it is not necessary to modify the definition of a superclass when introducing an (exceptional) subclass; since the process of stepwise refinement by specialization suggests that programming proceed by extending the class hierarchy at the bottom, we have achieved locality for such extensions.

Concerning the veracity of specifications: if looking at some class C we wish to find out what conditions apply to attribute p, the only additional information we need is the definitions of attributes which contain the clause **"excuses p on C"**.

We have provided a clear semantics for the "excuses" construct. It has the property that a constraint on some superclass is generalized when exceptional subclasses are encountered so that both the exceptional and unexceptional elements of the superclass satisfy the constraint. This permits the extent of subclasses to be included in the extent of superclasses, as desired.

We have also provided the embryo of an underlying type theory which can easily be *derived* from the specification of the classes, and which allows subclasses to be considered as subtypes of their superclasses, thus preserving bounded polymorphism.

Our solution is a "sanitized" version of the default inheritance mechanisms available in some Knowledge Representation languages, but which is extended to deal with contradictions arising in situations other than subclasses, as well as inherited integrity assertions (not just default values).

The paper also makes a secondary contribution to the field of semantic data modeling and conceptual modeling languages: It dissects and lays out clearly the various reasons for adopting the framework of classes and subclasses with inheritance as a basis for data and program organization. As a result of this analysis we are able to clearly separate the concept of "*class*", used for *descriptive* purposes, i.e., to build and maintain a description, from the associated "*type*", which is useful in type checking for detecting potential errors and in the optimization of queries. This separation turns out to be a key ingredient in our solution to the problem of exceptional (sub)classes. It also allows us to suggest a technique for defining range types for attributes which is missing in languages such as Taxis and Daplex: namely, the ability to define types of attribute structures without naming them, e.g., the "in-line" definition of Swiss addresses or of physicians certified by a certain board (***Physician [certifiedBy : {'ABO}]***). This avoids cluttering the schema with identifiers of classes which are of no interest, and saves the cost of maintaining their extents.

Acknowledgements. I have derived much benefit from discussions with participants of the Appin Workshop, whose proceedings, to appear as "Data Types and Persistence" (M.Atkinson, P. Buneman, R. Morrison eds, Springer Verlag), include some of the material motivating this problem.

[1] Nixon, B., L.Chung, D.Lauzon, A.Borgida, J.Mylopoulos and M.Stanley.
Implementation of a Compiler for a Semantic Data Model: experience with Taxis.
In *ACM SIGMOD '87 Proceedings.* May, 1987.

[2] Borgida, A., J. Mylopoulos and H.K.T. Wong.
Generalization/Specialization as a basis for Software Specification.
In Brodie, Mylopoulos and Schmidt (editors), *On Conceptual Modeling*, chapter 4, pages 87-114. Springer Verlag, 1984.

[3] Borgida, A.
Features of languages for the development of Information Systems at the Conceptual level.
IEEE Software 2(1):63-73, January, 1985.

[4] Borgida, A.
Language Features for Flexible Handling of Exceptions in Information Systems.
ACM Trans. on Database Systems 10(4):565-603, Dec., 1985.

[5] Brachman, R.
What IS-A is and isn't: an analysis of taxonomic links in semantic networks.
IEEE Computer 15(10), 1983.

[6] Buneman, P. and Atkinson, M.
Inheritance and Persistence in Database Programming Languages.
In *Proc. SIGMOD'86 Conference,* pages 4-15. May, 1986.

[7] Cardelli, L.
A semantics of Multiple Inheritance.
In *Proc. of Symp. on Semantics of Data Types.* Springer Verlag, Sophia Antipolis, France, June, 1984.
(Lecture Notes in CS #173).

[8] Cardelli, L., and P. Wegner.
On understanding types, data abstraction and polymorphism.
ACM Computing Surveys , 1986.

[9] Chan, A., S.Danberg, S.Fox, W.K.Lin, A.Nori and D.Ries.
Storage and Access structures to Support a Semantic Data Model.
In *VLDB '82 Conference Proceedings.* 1982.

[10] Chan, A., U.Dayal, S.Fox and D.Ries.
Supporting a Semantic Data Model in a Distributed Database System.
In *VLDB '83 Conference Proceedings.* 1983.

[11] Smith, J.M. and D.C.P.Smith.
Database abstractions: aggregation and generalization.
ACM Trans. on Database Systems 2(2):105-133, June, 1977.