# ACCOMMODATING EXCEPTIONS IN DATABASES, AND REFINING THE SCHEMA BY LEARNING FROM THEM

*Alexander Borgida*
*Keith E. Williamson*

*Department of Computer Science*
*Rutgers University*
*New Brunswick, NJ 08904*
*USA*

**Abstract**

To utilize DBMSs, a database designer must usually construct a *schema*, which is used to validate the data stored and help set up efficient access structures. Because database design is an art, and because the real world is irregular, unpredictable, and evolves, truly useful database systems must be tolerant of occasional deviations from the constraints imposed by the schema, including the semantic integrity constraints. We therefore examine the problems involved in *accommodating exceptional information* in a database, and outline techniques for resolving them.

Furthermore, we consider ways in which the schema can be *refined* to better characterize reality as it is reflected in the data encountered, including the exceptions. For this purpose, we describe part of a "database administrator's assistant" - a computer system which can suggest modifications and additions to the current definitions and integrity constraints in the schema. This system makes generalizations from the currently encountered exceptions, and is based on techniques used in Machine Learning.

## 1. INTRODUCTION

It is by now conventional wisdom to consider a database to be a *model* of some portion of the real world [Tsichritzis 82]. As with any other model, the usefulness of the database depends on its correctness and accuracy: A database which contains incorrect information is worse than having no database since it gives a false sense of security. And, obviously, if a fact is known, it is better to have it in the database: one can make better decisions with more information.



In databases, the above-mentioned world model is normally partitioned into two parts: i) the *schema*, which captures generic, time-invariant information; ii) the *facts*, which are specific, volatile and occur in large quantities. From the user's point of view, the schema describes what concepts the database knows about (e.g., the classes and properties of a semantic data model [Borgida 85]) and what constraints exist on the possible relationships between objects (e.g., integrity constraints of various kinds). This paper addresses the problems arising when the schema of a database does not model the respective aspects of the world entirely accurately.

In particular, we first present a technique for storing and accessing in the database information which does not conform to the constraints imposed by the schema, but which nonetheless reflects the correct state of the real world. We call such information *exceptional information*. The ability to accommodate such exceptional information allows a much greater degree of flexibility for the users of the database, since the database of facts can now be more accurate than before without the need to recompile the entire database system. In the second part of the paper, we follow this up by describing the principles of a program whose goal is to propose refinements to the integrity constraints and definitions of the database based on an analysis of the exceptions encountered so far. Such an algorithm can be viewed as part of a more general "database administrator's assistant": a program which makes suggestions on ways in which the database schema could be improved based on evidence of its use. Our work is based on, and extends, techniques for generalization from examples in the field of learning in Artificial Intelligence.

In the next section, we present a particular language for describing the schema of databases -- essentially a "semantic data model" -- and motivate the need for permitting violations to constraints to persist in a particular database. In section 3, we then introduce a mechanism which allows such exceptional information to be stored and accessed in the database. Section 4 describes some ways in which a computer system might aid in the refinement of a database schema based on the exceptions to it encountered so far, and the last section proposes a technique to accomplish this based on empirical generalization.

## 2. DATABASE CONSTRAINTS AND EXCEPTIONS

### 2.1 Conceptual databases and schemas

We have carried out our investigation in the general context of what are known as "semantic data models" (e.g., DAPLEX, SDM, TAXIS) [Borgida 85], although a similar treatment of exceptional information is possible in the framework of more traditional data models such as the relational one.

We see a database state as a collection of *object descriptions*: every object is an instance of one or more *classes*, and has 0 or more *attributes/properties*, which relate it to other objects, and thus describe it. Note that each object has a *distinct identity* independent of the values of its attributes (contrast this to the relational model, where a tuple's identity is defined by its key attributes).

The schema of such a database then specifies two things: **(1)**. The *definition of the classes* available in this database, including their name, the attributes applicable to each of their instances, the range of possible values for every attribute, and constraints on the membership of various classes (such as subclass relationships); this essentially defines a typing mechanism for database objects. **(2)**. Additional semantic integrity *constraints* (ICs) limiting the possible relationships in the database, expressed in some sort of logical language.

Before proceeding with the details of our specific data model, it may be worth briefly reviewing here the purpose of the schema:

1. From the user's point of view, the schema: a) describes the "domain of discourse" of the database; and b) checks the correctness of data entered.

2. From the DBMS point of view, the schema aids in achieving various forms of efficiency: (a) *storage efficiency*: e.g., by allowing concise codes to be assigned to enumerated types; (b) *faster retrieval*: e.g., fixed-length record schemes can be used for storage when attributes and their ranges are known; or the subclass hierarchy can be used to suggest vertical and horizontal splitting strategies [Chan 82]; (c) *compile-time optimizations of programs*: by using type information, one can often eliminate run-time type checking; (d) *semantic query optimization*: integrity constraints can be used to find faster access paths to the data [King 80].

We will introduce the language of class definitions used in this work through an example, relying on the reader's ability to generalize. According to the definition in Figure 2-1, the class of EMPLOYEE objects is a subclass of PERSONs (i.e., every EMPLOYEE instance is also a PERSON instance) with attributes **name**, **degree**, **jobCat**, **wages** and **supervisor**. Furthermore, **names** of employees are strings of no more than 25 characters, **degree** can be one of HSGD, BS, MS or PhD, **jobCat** is a integer between 1 and 7, **wage** is a decimal number in the range 0.00 to 80000.00, and the **supervisor** of an employee must also be an employee.

Note that as usual in semantic models, classes "inherit" the attributes of their super-classes; hence, if PERSONs have **age** and **address** attributes, then so will EMPLOYEEs. In addition, when defining a subclass one can also specify a restriction on the range of an existing attribute; for example, PERSONs may have had **age** defined to have range 0..130, but for EMPLOYEEs this may be restricted to be **14..90**.

```
EMPLOYEE == PERSON with
        [name : STRING(25)]
        [degree : {'HSGD, 'BS, 'MS, 'PhD}]
        [jobCat : 1..7]
        [wages : 0.00 .. 80000.00]
        [supervisor : EMPLOYEE]
```

**Figure 2-1**: Definition of EMPLOYEE class

The definition of a class such as EMPLOYEE specifies *necessary* properties for objects belonging to that class, but it is up to the user to explicitly assert that an object is an instance of it (i.e., there is no recognition of objects). However, in some cases we need classes for which we have sufficient conditions determining membership. For example, secretaries are by definition exactly those employees who have job category 6. Such classes are described by selecting some "definitive" property specifications (in the above case, **jobCat** being 6) as tests applicable to a *base class* (EMPLOYEE, in the above case):

```
SECRETARY == EMPLOYEE
        such that
                [jobCat: 6..6]
        with
                [salary: 0.00 .. 20000.00]
                [typingSpeed:  20..80]
```

Note that test-defined classes may also have additional necessary conditions attached to them.

A database state then records information about the current state of the world by keeping track of the membership of objects in classes, and the known property values of objects. For example, we may have the following information about an object emp45 at some moment:

```
EMPLOYEE <emp45>
        ([name="karl marx"] [age=88] [jobCat=4])
```

leaving it to be inferred from the subclass hierarchy that emp45 is also a PERSON.

The language of class definitions allows us to capture certain constraints on the possible facts in the database. Most obvious are the *property constraints* limiting the possible ranges of properties. As mentioned earlier, *integrity constraints* are used to enforce additional consistency conditions not captured by the restrictive language of class definitions. ICs will be stated in a first-order language where the attribute names are considered as functions, class names are unary predicates and the primitive predicates are $=$, $\neq$, numeric and string comparators. For example, the following two constraints capture part of the semantics of the terms introduced in the above definition of class EMPLOYEE:

$(\forall x)$ EMPLOYEE(x)$\Rightarrow$(x.supervisor$\neq$null)
$(\forall x)$ EMPLOYEE(x)$\Rightarrow$(x.wages$<$x.supervisor.wages)

### 2.2 Exceptions in databases.

In most practical situations we will find that the schema developed during database design does not perfectly describe the world in the sense that there will occasionally be information which we want to store, but which contradicts the constraints of the schema. One obvious reason for this is that the world has changed since the database was designed. Another reason is that database design is an art, not a science, often pursued by someone unfamiliar with the

application domain: hence the schema may simply be wrong. A third reason is the *variability of the real world*. It is often neither feasible to anticipate, nor desirable to capture all possible situations that may occur in the world. Philosophers have long been aware of the problems posed by "natural kinds" -- concepts which occur in our everyday experience rather than being defined. Unfortunately databases often hold information exactly about such natural domains. The following examples illustrate some of the problems involved:

- A few persons may get paid more than $80,000.00, thus violating the constraint on **salary**; unfortunately, the upper bound will always be debatable (100K?, 1000K?), and if we use such a high bound, we'll lose much of the error checking function of the constraint.
- Some employee may have received a foreign degree (e.g., a French "Baccalaureate" or 'Bac), which is not even closely equivalent to any American one, yet decisions such as promotions, raises, etc. are based partly on this; it is therefore important to keep the actual value of the degree in the database.
- Because of work on a new classified project, a few employees may receive security clearances of various degrees. The company then desires to record this information in the database for possible future references.
- In some special cases, a person may in fact earn more than his/her supervisor.
- Although it is reasonable to describe the class of valid addresses for a country like the USA, it seems impractical, if not impossible, to describe the addresses for *all* the countries in the world just in case we encounter someone from that country. At issue here are the necessary attributes for an address, not just their legal values.

Therefore databases need the flexibility to accommodate special cases at runtime that are not sanctioned by the schema.

## 3. HOW TO LIVE WITH EXCEPTIONS

There are a number of problems which arise when we consider allowing the constraints of the schema to be violated. The following are some of the areas of concern:

1. *Storage and Access Efficiency*: The most obvious, though not the most interesting, problems arise when we consider the implementation of the DBMS. When inherent constraints of the data model are violated, we must be prepared to store extra attributes, to find attribute domain-constraints violated, non-unique keys, etc. all of which are problematic for most traditional techniques of mass data organization, based on records. Note that we don't want the presence of exceptions to degrade the efficiency of handling normal data, though we will assume relatively few exceptions occur overall.
2. *Semantics of computations:* Exceptional data will need to be treated with circumspection in computations, since its semantics may be different. For example, if the supervisor of an employee turns out not to be an employee (she may be a consultant, say), then she may not have some attributes normally expected of all employees (e.g., **jobCat**).
3. *Sharing:* The same problems arise in the interpretation of data when we realize that there are several users or programs utilizing the database. For example, a user should be warned that an attribute which normally holds an integer representing US Dollar values now has Swiss Francs in it.
4. *Administration and accountability*: As with other updates, there is an obvious need to control the ability to allow exceptional facts to enter the data base.
5. *Validation of future updates*: Although violations of semantic integrity constraints do not usually pose storage/retrieval problems, they do cause logical difficulties: If an integrity constraint is violated in a database, and we allow the violation to persist because it is a special case, then the constraint will henceforth be inconsistent with the database. This means that the IC will not be able to detect errors in more recent updates since it will always be "ringing the alarm", and we will not be able to tell whether it is just a false alarm due to the old exception.

To resolve some of these issues, observe that our database can store facts in two essentially distinct ways: through the membership of objects in classes and through property values for objects. In this paper we will treat only exceptional property values, though exceptional instances of classes can be dealt with in a parallel manner. Let us assume that facts about property values are entered into the database by two primitive operations, **modify** and **createObject**, and are retrieved by the operator **getValue**.

One simple way of storing and retrieving exceptional property values -- ones which lead to the violation of constraints imposed by the schema -- is to have a second set of operators: **exnal_modify**, **exnal_createObject** and **exnal_getValue**. These operators will take care of the special ways in which exceptional facts must be stored, given that the schema has been used to optimize storage and access structures for the normal case. For example, while ordinary facts will be stored according to standard techniques (e.g., [Chan 82]), exceptional facts will be stored in a separate (logical) file of records, with variable length fields, each of which holds a single fact (e.g., the object, property name, property value, type information about the value, etc.). This second file is then accessed and modified by the **exnal** operators, and can be indexed and clustered to improve efficiency.

To deal with the next two issues, related to semantics and sharing, we adopt the philosophy that users/programs must be *cautioned* when they encounter exceptional data, so that they can determine whether normal procedures apply here, or whether special actions need to be taken. To accomplish this, we adopt a uniform framework for signalling both the initial violation of constraints, which is necessary in any case, and the use of exceptional property values. This framework is based on the concept of "exception signalling and handling" common in modern programming languages such as ADA™, and originally described by Goodenough [Goodenough 75]: Whenever an error or special situation arises, such as division by 0, an operation is aborted and an *exception* is signalled; the program (or user in our case) can then propose a *handler*: a fragment of code which is executed instead of the interrupted operation. In our case, an update or retrieval operation will signal that a constraint violation has been detected by creating an object in the special class VIOLATION. This violation object should carry information about exactly what has gone wrong. To accomplish this, every constraint is named, and then the violation records this information through its properties. For example, if croesus'

degree property is being updated to 'Bac by the operation

**modify**(croesus,degree,'Bac)

then the violation object would identify the constraint being violated, namely [degree : {'HSGD,'BS,'MS,'PhD}], as (EMPLOYEE,degree).

Usually, such a violation would uncover an error which would need to be corrected; but if the value is correct, we can force it to be actually stored by invoking **exnal_modify**(croesus,degree,'Bac) in the violation handler. This operator in addition marks the respective property of that object as requiring special handling. This is accomplished by defining a special class of objects

EXCEPTIONAL with
[prop : PropertyIdentifier]
[obj : AnyObject]
[class: ClassIdentifier]
[withRespectTo: AssertionIdentifier]

whose instances mark exceptional properties of objects. We require that such an instance be created as part of exceptional updates. Thus, in the above example **exnal_modify** would also have taken as an argument the violation (which held the identifier of the assertion being violated), and in addition to storing croesus' degree, it would have created the object e1, shown below.

EXCEPTIONAL <e1>
( [prop=degree] [obj=croesus] [class=EMPLOYEE]
[withRespectTo=(EMPLOYEE,degree)] )

Objects in the class EXCEPTIONAL can be examined by database users to find, for example, all exceptions to certain rules or all exceptions involving certain properties. Thus the `obj` property values of objects satisfying the query

EXCEPTIONAL(x) ∧
x.prop=degree ∧ x.class=EMPLOYEE

would locate all employees with exceptional degrees. This is then one technique whereby users sharing the database, including administrators, can become aware of exceptions introduced by others.

More importantly, when someone tries to retrieve a property value which turns out to be exceptional, the DBMS raises the corresponding EXCEPTIONAL object as a violation - in some sense an "echo" of the original violation. For example, if we now try to retrieve croesus' degree using **getValue**(croesus, degree), then e1 is raised as a violation. The user can then invoke **exnal_getValue** in the violation handler to obtain the actual stored value, 'Bac. After seeing this value, the user could decide to continue with the current plan or take special measures.

Note that the above mechanism can also be used to store new attributes - ones not mentioned in the schema. For example, one can record the security clearance code of a particular employee by the operation **exnal_modify**(croesus,securityCode,10).

We would like to make the treatment of exceptions to ICs resemble that of exceptions to property constraints. [1] Therefore we will label ICs and attach them to classes: the assertion concerning the supervisor's salary

notOverManager (∀x) EMPLOYEE(x)
(x.wages < x.supervisor.wages)

could thus be identified as (EMPLOYEE. notOverManager). Violations of ICs are then signalled in an identical manner to those of property range constraints, and the user may "blame" this on the value of zero or more properties, which are marked exceptional. For example, if the above constraint fails when x=croesus then any of the following facts may be exceptional: croesus.wages, croesus.supervisor or croesus.supervisor.wages.

Finally, consider the problem of continued integrity checking: if after an update croesus earns more than his supervisor, then the constraint `notOverManager` will always be false as long as his and his supervisor's salary do not change. We therefore propose to ***modify*** the IC so that this "false alarm" is avoided, and would like to make this modification "minimal" in the sense that other errors which would have been caught by the original IC will continue to be detected by the new constraint. In this paper we will adopt the relatively straightforward approach of considering every IC to be actually of the form:

$$constr_i : (\forall\omega)\ \mathrm{SPECIAL}constr_i(\omega) \vee \Phi(\omega)$$

where $constr_i$ is the label of the rule, $\omega$ is a sequence of variables, $\Phi$ is the original form of the IC, and $\mathrm{SPECIAL}constr_i$ is a predicate which prevents the actual condition from being evaluated for special cases. Initially, $\mathrm{SPECIAL}constr_i$ is everywhere false, but as exceptions are encountered and "excused" for various argument tuples $\delta_1$, $\delta_2$, ..., the definition of $\mathrm{SPECIAL}constr_i$ becomes

$$\mathrm{SPECIAL}constr_i(\omega) \Leftrightarrow \omega=\delta_1 \vee \omega=\delta_2 \vee \ldots$$

Thus, after encountering croesus as an exception, the constraints concerning managers' salaries would actually look as follows:

(∀x)(SPECIALnotOverManager(x) ∨
(EMPLOYEE(x)⇒(x.wages<x.supervisor.wages)))

SPECIALnotOverManager(x) ⇔ (x=croesus)

In summary, we have proposed to accommodate exceptions to constraints in databases by 1) marking exceptional information using objects in the database so that information *about* exceptions can be maintained, 2) using an exception handling mechanism to alert the user when exceptional information is being manipulated, as well as when constraints are violated, 3) providing special operators to do this manipulation, and 4) modifying integrity constraints so that they are consistent with the exceptional facts. For the interested reader, a considerably more general and complete proposal for exception handling is presented in [Borgida 84]. This includes considerations about software engineering issues, implementation, accountability, and techniques for dealing with transactions. In addition we provide a more refined theory of how to deform ICs in a "minimal" way, including model and proof-theoretic accounts of this phenomenon.

## 4. LEARNING FROM EXCEPTIONS

The way that a user determines whether the schema of a database is ill-designed is to *see it in use*: to watch the queries and updates issued against it. Note that as long as everything entering the database must be shoe-horned to fit the schema, there is no way for the *system* itself to determine that the schema is somehow wrong. However, once we allow the database to accommodate exceptional facts, then the

[1] Note that property constraints of the form `[jobCat: 1..7]` are equivalent to quantified formulas like

(∀x) (EMPLOYEE(x) ⇒ 1≤x.jobCat ∧ x.jobCat≤7).

system can itself identify cases in which the number of exceptions mounts to the point where a change in the schema is indicated. We wish to investigate here the possibility of having automatic aids which watch for such occurrences and then suggest to the database administrator possible improvements to the schema. In this sense we are describing a part of an expert system which we might call a "database administrator's assistant".

Given this goal, the first question is what kinds of changes to the schema are we contemplating. We will restrict our attention to incremental modifications of the following sort:

- *Modifying ICs to apply in more restricted circumstances.* Thus "all employees earn less than their supervisors" could be qualified with "unless their supervisor's status is part-time." Such qualification takes place by defining the predicates SPECIALc occurring in all constraints in a more general way than just by enumeration, for example by describing a new class whose instances include all the exceptions.
- *Adding new attributes to a class definition.* For example, a `securityCode` property could be added to the employee class.
- *Defining a new class and placing it in the current subclass hierarchy.* An alternative to adding the `securityCode` property directly to the EMPLOYEE class would be to create a subclass CLEARED_EMPLOYEES of EMPLOYEES, with property `securityCode`, among others. A different example: if all employees who earn over 80K (and hence are exceptions to the salary range constraint (EMPLOYEE,salary)) have jobCat 1 or 2, then perhaps a subclass UPPER_LEVEL_MANAGEMENT of employees should be created for them.

Of course there are many other possible changes that one might want to make to a schema (dropping attributes, changing their names, making ICs stronger or adding new ones, etc.) but the information present in exceptional facts seems to support mainly the changes listed above.

The following are some of the benefits of the above mentioned changes:

1. Fewer exceptions will need to be excused in the future. As we have seen, introducing and accessing exceptional facts involves the overhead of handling violations and excuses for them, so this gain can be quite significant.
2. Refined ICs may be more useful for semantic query optimization, and even evaluating ICs may be faster: checking a single attribute value may be faster than searching through a sequence of disjuncts to see if any apply.
3. The addition of new attributes and classes gives the user a refined vocabulary for expressing queries and updates.
4. The increased vertical/horizontal splitting provided by new classes in the hierarchy might be used to improve retrieval efficiency .

We present next a technique for characterizing the class of currently known exceptions, and thence adjusting the schema to accommodate them. This technique locates commonalities for sets of objects based on their class memberships and property values, with the limitation that these commonalities must be expressible in the language of schema definitions.

# 5. USING EMPIRICAL GENERALIZATION FOR SCHEMA REFINEMENT

## 5.1. The utility of empirical generalization.

Suppose we have an algorithm Descr which given a set of objects {a1,a2,...} infers a class description for them. In particular, it provides the definition of the/a most specific class which contains all these objects, given the language of class definitions from Section 2. Thus, Descr will output class definitions that consist entirely of "definitive" attribute specifications attached to some least general existing class which contains these objects as instances.

We can use Descr to accomplish some of the goals stated earlier as follows:

1. *Improving constraints.* If for some constraint $c$, the predicate SPECIALc is defined as

   $$\text{SPECIAL}c(x) \Leftrightarrow x=e_1 \vee x=e_2 \vee \ldots \vee x=e_n$$

   as a result of a number of exceptions $e_1$, $e_2$, ... $e_n$, then replace this definition by

   $$\text{SPECIAL}c(x) \Leftrightarrow \text{Descr}(\{e1,e2,\ldots\})(x)$$

   Since class descriptions cannot consist of enumerations of instances, the predicate SPECIALc will typically be generalized to admit more objects as implicitly exceptional, without the need for user intervention.
2. *Adding new attribute specifications.* Suppose that some new attribute `attr`, which does not appear in the schema, has been introduced in several exceptional facts $a_1$.`attr`=$v_1$, ... $a_n$.`attr`=$v_n$. Then we may want to propose that the attribute specification [`attr` : Descr($\{v_1,\ldots,v_n\}$)] be added to the least class containing $a_1,\ldots,a_n$.
3. *Changing attribute specifications on existing classes.* If for some class C, which currently has property definition [p : E], there are many exceptional property values $a_1.p=v_1$, ..., then find the most general existing subclass(es) C' of C to which the constraint [p : E] can be restricted (i.e., such that C' does not contain any of the exceptional objects $a_1,\ldots$); if the constraint on p has not been refined between C and C' then we can augment the definition of C' with [p : E], and then generalize the constraint on C to be [p : Descr(E $\cup$ $\{v_1,\ldots\}$)].

Introducing new classes is of course another way to modify the schema. In general, the decision to introduce a subclass is a heuristic one, classes being defined in order to represent collections of objects to which certain properties are restricted, and/or collections over which ICs or queries may be quantified. Specifically, in each of the cases in the above list, there are several classes which may be of use if added to the database schema:

1. While modifying constraints, there is evidence for a new class Descr({e1,...}) describing the objects to which the constraint did not apply.
2. While adding a new attribute specification, there is evidence for the existence of the class describing objects to which the attribute applied, Descr($\{a_1,\ldots\}$), as well as the set of values for the new property, Descr($\{v_1,\ldots\}$).
3. When an attribute specification is modified, there is evidence for a class to which the old definition applied, Descr(C - $\{a_1,\ldots\}$) as well as a more general class for the

range of that property $Descr(E \cup \{v_1,...\})$.

We realize that there is in fact a tradeoff between adding new classes to make it more convenient to communicate with the database, and the confusion caused by the presence of too many marginally useful class definitions. We propose to protect the DB manager from a blizzard of suggestions for class definitions by using a scheme in which suggestions for class definitions that appear to be useful are placed in a "suggestion box", and a second program sifts through these to find evidence for truly useful definitions, sometimes as an amalgam of several individual suggestions. The details of this algorithm are currently under investigation.

In summary, we see the following scenario: The DB administrator's assistant keeps track of violations and exceptions to constraints in the schema. When prompted by the DB administrator or when sufficient evidence is accumulated[2], it suggests one or more alternative changes to the schema to be carried out. The DB administrator has the final decision about which changes, if any, to implement. In fact, the administrator may chose to modify the class suggestions by, for example, removing or relaxing certain property restrictions (since they were the result of coincidences in the particular sample of exceptions), or by making some property constraints necessary conditions only, rather than part of the test for class membership.

We present next the basic algorithm underlying Descr which we have used, and in the following section we describe a number of refinements to improve it.

### 5.2. The generalization algorithm

Descr takes as input a set of object descriptions, and produces a class description which has as instances these objects. One of our goals is to make this the "most specific" such class, in the sense that any other class which would have them as instances would subsume this class. One reason for this is that we do not want to over-generalize constraints, since their goal is to detect errors: a most specific class makes the SPECIAL predicate apply to the fewest cases. We shall first describe how we approach this goal, and then deal with other, possibly conflicting goals for Descr, in the following subsection.

The fundamental idea of the algorithm is to consider the objects to be described $a_1$,$a_2$, ... one by one, and in each case deform the current class description by the least amount necessary to have it also describe the current object. The description of the set of exceptions is initialized to be the first exceptional instance. The next exceptional instance is then compared against this. Whenever part of the class description is too restrictive, that part is generalized by one of the rules below. In the technical terminology of machine learning, the algorithm Descr makes a specific-to-general breadth-first search [Mitchell 82, Michalski 83] to induce a description for a set of objects.

1. *Introduce/Expand Range* - A range of values can be introduced or expanded to cover a new scalar data value (string, number, enumeration). For example, to describe salaries of 40k and 60k, the attribute specification [salary: 40000..60000] will be necessary. In the case of strings the generalization will be based on the length of the strings.
2. *Introduce/Generalize Class* - An existing class can be introduced or generalized (using the IsA hierarchy) in a description to cover a new data value, especially an entity object. For example, [supervisor : ACCOUNTANT] can be generalized to [supervisor : EMPLOYEE], if the next example's supervisor is not an accountant. In this generalization process, one uses the least class in the hierarchy of classes which subsumes both the old class and the new value. We note that this generalization could also be applied to scalar values if our language allowed classes of scalars to be defined, such as EMPLOYEE_AGES.
3. *Drop Attribute Specifications* - Descriptions are prunned by dropping attribute specifications which are not uniformly applicable. For example, in generalizing a description from "EMPLOYEE such that ..." to "PERSON such that ...", we must eliminate references to attributes which are specific to EMPLOYEEs but do not occur on PERSONs. Specifications are also dropped when they are redundant or vacuous, such as in "EMPLOYEE such that ... [degree : {'HSGD,'BS,'MS,'PhD}]."

For example, suppose we want to find a description for the following 3 employees:

RESEARCHER <e41>
([age=35] [degree=PhD] [jobCat=2] [wages=85k] [supervisor=EMPLOYEE<e56>])
RESEARCHER <e57>
([age=45] [degree=MS] [jobCat=2] [wages=90k] [supervisor=EMPLOYEE<e56>])
EMPLOYEE <e66>
([age=50] [degree=PhD] [jobCat=1] [wages=95k] [supervisor=EMPLOYEE<e78>])

The initial description would be

[RESEARCHER such that
[age : 35] [degree : 'PhD]
[jobCat : 2] [wages : 85k]
[supervisor : EMPLOYEE<e56>]]

The second researcher e57, forces this description to be generalized to

[RESEARCHER such that
[age : 35 .. 45] [degree : 'MS .. 'PhD]
[jobCat : 2] [wages : 85k .. 90k]
[supervisor : EMPLOYEE<e56>]]

The third employee e66, forces this description to be generalized to

[EMPLOYEE such that
[age : 35 .. 50] [degree : 'MS .. 'PhD]
[jobCat : 1 .. 2] [wages : 85k .. 95k]]

The description of supervisors was generalized to [supervisor : EMPLOYEE], since e66 has a different supervisor than the other two employees. This restriction was then dropped since supervisor's of employees are always employees.

### 5.3. Refinements to the generalization process.

In this section we present a number of ideas designed to resolve certain problems with the algorithm presented in the previous subsection.

To begin with, in the schema there may be more than one maximally specific class that contains an object. For example, a person may be an employee and a customer at

[2]The decision on when enough evidence has been accumulated to suggest changes is heuristic, and may be based on absolute or relative numbers of exceptions, and the "goodness" of the induced description, which measures how useful or precise the description is.

the same time. Also, in general, the subclass relationship need not form a tree, so that two classes may have more than one most specific subsumer in the IsA hierarchy. For these reasons, there may be more than one most specific description for a set of objects. Therefore, Descr must actually maintain a *set* of maximally specific descriptions (as is done in [Hayes-Roth 76, Vere 78]). This set is initialized to be the minimal classes which contain the first object to be described. Given another object, Descr generalizes each description in the set as described earlier. In "climbing" the generalization hierarchy, the set of most specific descriptions may grow if there is more than one parent. On the other hand, when one description in the set is generalized, it may now subsume one of the other descriptions and hence is no longer necessary. In this case, the set of most specific descriptions may shrink.

A more important source of problems is an inherent conflict between some of our goals for finding the generalization. On the one hand, we have looked so far for the *most specific* description of a set of exceptional objects, because this minimizes the set of objects for which the original integrity constraint is not checked, and hence allows us to continue to detect errors effectively. In finding the most specific description, the algorithm Descr may generate spurious restrictions based on accidental commonalities in the data, unless we have a very large sample of exemplar objects. For example, a subclass of employees is likely to have few PhD's among them, and hence the restriction [degree: 'HSGD..'MS] may be suggested, though it is not necessarily relevant to characterizing this set. Also, Descr may introduce circular reasoning along the lines "the employees who earn over 80K are those who satisfy the constraint [salary : 85000..120500]". The presence of such spurious attribute specifications is bothersome for two reasons: it makes integrity constraints more expensive to check, and it allows fewer objects to be special, which may circumvent our goal of eliminating the need to excuse exceptional cases. These two problems can be resolved by selectively eliminating all but the most "essential" attribute specifications in a description. Of course, if we drop too many property restrictions or the wrong ones, the constraint expressed in the schema will become over-generalized, thereby making it less effective for detecting errors. We will therefore make use of a number of heuristics in order to decide which attribute specifications can or cannot be dropped from a description.

To avoid over-generalizing a constraint, we can try looking for *negative examples*: objects which should NOT be covered by the class generalized by Descr from a list of examples. Since our goal is to retain the ability to detect errors, the best candidates for negative examples are those circumstances where the current constraint did detect a violation that was NOT excused - a real data entry error. In other words, a constraint can be considered over-generalized if it no longer detects some of the errors it caught earlier. Therefore the generalization of the predicate SPECIAL should cover as few such negative examples as possible. Negative examples allow us then to chose between alternative descriptions of a set of exceptions to a constraint, and to decide when dropping some attribute specification from a description is ill-advised.

If the number of entities in the database which satisfy a description increases dramatically when some attribute specification is dropped, then the resulting description has a good chance of being overly general; this provides a second global heuristic for determining when an attribute constraint is essential. We are not currently using this heuristic, preferring to rely on statistical information on the distribution of values in the database, as explained below.

This still leaves us with the problem of deciding which constraints to try to drop in a description, since it may not be feasible to try all combinations, and since we may not have enough positive and negative examples to work with.

For this purpose we introduce the notion of *"relevance"*. In any situation we wish to have an evaluation of which attribute specifications are more relevant to the generalization at hand so that irrelevant ones can be discarded. The following are some heuristic sources of relevance.

- Classes, and their properties, which are closer to a particular class in the superclass hierarchy are more relevant to it than ones further away. In particular, the new attributes introduced when a subclass is defined would seem to be most relevant to each other and to that subclass.
- If we are using Descr on values violating an IC, then the other classes and properties mentioned in the IC are likely to be relevant.
- If we are describing objects which are exceptional with respect to the range constraint of some attribute p, then their p property is irrelevant to this characterization. (This prevents circular descriptions.)
- The availability of information about the distribution or frequency of certain values in the database can also be used in deciding relevance. For example, if certain values of a property p are statistically infrequent in the database (e.g., 'PhD degrees) then the absence of those values in a restriction is not very significant (so the constraint [degree: HSGD..MS] could be dropped). Conversely, if a value of a property occurs very frequently in the database (e.g., jobCat 6 or 7, representing low-ranking employees) then the absence of these property values in a set of examples is quite significant (so [jobCat: 1..2] would seem to be a relevant constraint). These intuitions will eventually be made more precise through the proper application of statistical techniques which establish whether there are significant differences between the distributions of property values for the general population and the set of exceptional objects.

As an example, if the description learned at the end of the last section was being used to describe employees who violated the constraint [**wages**: 0.00 .. 80000.00] on EMPLOYEE, then the following description would be more appropriate.

EMPLOYEE such that
[degree : 'MS .. 'PhD]
[jobCat : 1 .. 2]

Two attribute restrictions were dropped. The **age** attribute is less relevant than **degree** and **jobCat** since it was introduced for PERSONs and not specifically for EMPLOYEEs. The **degree** and **jobCat** restrictions were also kept because they were statistically significant. On the other hand, the **wage** attribute is irrelevant to characterizing independently employees with high wages, so it was dropped. Note that in order to avoid dependence on the order of examining the examples during the generalization process, the pruning of descriptions should be delayed until after the algorithm Descr completes.

In order to combine the suggestions of the various sources of information about the merits of dropping some attribute restriction from a description we use a weighted sum of the

individual estimates. The exact numbers for this computation, as well as the thresholds to be used for cut-offs can only be determined empirically. We have therefore implemented most of the learning algorithm described above as a prototype PROLOG program, and we plan to carry out experiments with it to determine, among others, the appropriate definition of relevance. We also continue to refine and look for new heuristics for determining relevance.

A final problem to be addressed here concerns the restrictive nature of the generalizations possible. Techniques of empirical generalization are in general limited by the language chosen for describing generalizations. In particular, although our language allows characterizing commonalities of objects in terms of their properties, it does not allow referring to properties of their properties, for example. Thus, if persons have addresses, which are objects that have properties such as `street`, `city`, `state`, etc., then we are unable to characterize the set of persons living in New York City.

We can remedy this by extending the language to allow nested class definitions as a way of imposing additional integrity constraints[3]. For example, the following definition ensures that fathers of persons are male:

PERSON with
[sex : {'male, 'female}]
[father : PERSON such that [sex : {'male}]]

At this point, we can add another entry to the list of ways whereby a description is generalized in Descr:

- *Introduce Attributes* - A reference to a particular instance of a non-scalar class can be generalized by introducing a description of its attributes. For example, [supervisor : EMPLOYEE<e14>] might be generalized to also describe [supervisor : EMPLOYEE<e21>] by the attribute-specification

  [supervisor : EMPLOYEE such that
  [degree : 'BS] [jobCat : 3..4] [wages : 50k..60k]
  [supervisor : EMPLOYEE<e43>]]

  in case they both have e43 as supervisor and 'BS degrees.

This generalization can be accomplished by a recursive call to the Descr procedure on the values of the respective property (`supervisor` in the above example).

Of course, unless we are careful, this can lead to a sudden growth in the size of the descriptions, which is not desirable in light of the second and third goals for generalization mentioned earlier. Worse, it can lead to infinite recursion, as in the case when persons have spouse attributes, whose values are also persons. The solution to this problem lies in the judicious use of relevance: clearly, the farther we get away (in terms of properties of properties) from the original set of exceptional objects, the less relevant things get, so that we will want to stop after a few levels. Also, in describing properties of properties, we should be more willing to drop attribute specifications which are not sharply restrictive, for example. For this purpose, the relevance threshold can be raised with each level of recursion.

The current generalization program can be easily modified to the case when the data model allows *multi-valued properties*, as in DAPLEX [Shipman 81] for example, since in that case our object descriptions would simply have the form

"EMPLOYEE<e78 (.. supervisor=e12 supervisor e75 ..)"

We are also extending the algorithm to be *incremental* so that the effort of learning a description for a set of exceptions at one point will not be lost when another description is required later on, after some more exceptions have been found.

[3]There are of course many other possible extensions to the generalization language, but this one seems most consistent with the spirit of the enterprise so far.

## 6. SUMMARY

One of the hallmarks of intelligent human behavior is the ability to cope with situations which represent deviations from the norm, and we would like for DBMSs to have this capability. On the other hand, we find ourselves in a situation where we wish to impose constraints on the data to be entered in the database both for detecting the ubiquitous data-entry errors, and for capturing regularities which are needed by the currently used techniques for managing large amounts of data efficiently. This paper has presented a technique for accommodating exceptional information in the context of a semantic data model. The technique is based on the following four principles:

- special information can be stored and manipulated by special operators in ways which differ from those used to deal with the large majority of normal data;
- since the database is shared, users must be warned when they deal with exceptional information so they become aware that something irregular has happened before;
- the ability to detect violations of constraints and the use of exceptional values can be conveniently captured within the framework of "exception handling" as this concept appears in most modern programming languages;
- integrity constraints must be refined to disregard the contradictions raised by exceptional values that have already been "excused", so that they can continue to be useful in detecting errors in new updates.

A second characteristic of intelligent human behaviour is the ability to adapt to change or to recognise and correct erroneous assumptions. In our case, exceptions of certain kinds may become quite common if the database schema was improperly designed to begin with, or if the application domain has changed slightly since the original design. In either case, we believe that computer tools can be used to redesign the schema, based on the exceptions encountered so far. In fact, we presented some of the principles of an experimental program which uses techniques of Machine Learning to solve parts of this problem. This technique attempts to find commonalities among the elements of a set of objects, such as exceptions to a rule encountered so far, and characterizes these in the form of a class description. Such a characterization can be used to adjust the database schema in various ways such as changing restrictions on property value ranges, modifying integrity constraints, and defining new classes. At the moment, we see this program only as an assistant to the database administrator, who makes final decisions about the appropriateness of new definitions, their naming, etc.

Although the approach described here is relatively efficient and robust, to the extent that it always produces some (better or worse) generalization, we are aware of several inherent shortcomings. These include the heuristic nature of the relevance function defined here, and the inability to handle negation and disjunction in descriptions. For these reasons we are currently investigating a second approach which uses a single exemplary exception to guide the search

for an *explanation* of why the constraint was violated - an explanation based on some underlying logical theory of the domain. This explanation could then be generalized to cover other potential exceptions which fit the same pattern, along lines similar to [Utgoff 84] or [Fikes 72].

Finally, we are currently investigating other ways in which information about the current use of a database can be used to improve its schema, such as looking at the queries issued againt it. These techniques would also be part of the proposed "database administrator's assistant".

**Acknowledgements:** Dr. Tom Mitchell, who is our collaborator in extensions of this work, has made many suggestions which have significantly improved this paper, and we are very grateful for them. This research has been supported by the National Science Foundation under grant No. MCS-82-10193, and Rutgers CAIP.

## REFERENCES

[Borgida 84] Borgida, A. "Language features for flexible handling of exceptions in Information Systems.", Technical Report LCSR-TR-70, Rutgers University, August 1984 (revised March 1985). (Submitted for publication).

[Borgida 85] Borgida, A. "Features of languages for the development of Information Systems at the Conceptual level." *IEEE Software 2(1)*:63-73, January, 1985.

[Chan 82] Chan, A., Danberg, S., Fox, S., Lin, W.K., Nori, A. and Ries, D. "Storage and access structures to support a semantic data model." In *Proc. 1982 VLDB Conference.* September, 1982.

[Fikes 72] Fikes, R., P.Hart and N.Nilsson. "Learning and executing generalized robot plans." *Artificial Intelligence 3(4)*:251-288, 1972.

[Goodenough 75] Goodenough, J.B. "Exception handling: Issues and a proposed notation." *Commun. ACM 18*:683-696, December, 1975.

[Hayes-Roth 76] Hayes-Roth, F. "Patterns of Induction and Associated Knowledge Acquisition Algorithms." In Chen, C. (editor), *Pattern Recognition and Artificial Intelligence.* Academic Press, New York, 1976.

[King 80] King, J. "Intelligent Retrieval Planning." In *Proc. 1st Natl. Conf. on AI*, pages 243-245. August, 1980.

[Michalski 83] Michalski, R., Carbonell, J. and Mitchell, T. *Machine Learning: An Artificial Intelligence Approach.* Tioga Publishing Company, 1983.

[Mitchell 82] Mitchell, T. "Generalization as Search." *Artificial Intelligence 18(2)*:203-226, March, 1982.

[Shipman 81] Shipman, D. "The functional data model and the data language DAPLEX." *ACM TODS 6*, March, 1981.

[Tsichritzis 82] Tsichritzis, D. and Lochovsky, F. *Data Models.* Prentice Hall, 1982.

[Utgoff 84] Utgoff, P. E. "Shift of Bias for Inductive Concept Learning." PhD thesis, Rutgers University, October, 1984.

[Vere 78] Vere, S. A. "Inductive learning of relational productions." In Waterman, D. A. and Hayes-Roth, F. (editors), *Pattern-Directed Inference Systems.* Academic Press, New York, 1978.

## APPENDIX: Algorithms

In order to define more precisely our ideas, we include here a pseudo-code description of our current algorithms.

*(i) The language for describing generalizations:* Classes whose instances are scalars (integers, enumerations, or strings) are described by the term **range**(lower-bound,upper-bound), where the bounds are either integers or enumeration elements. For strings, any ordering relation (e.g. $<$, $\leq$) refers to the lengths of the strings, so that, for a collection c of strings, *min*(c)=the length of the shortest string in c, *max*(c)=the length of the longest string in c, and thus c is described by **range**(min(c),max(c)). Dataclasses are classes whose instances are described by attribute values. Restrictions on dataclasses are described by constraining these attributes. The terms for describing a set of dataclass instances conform to the syntax:

bound-description ::= **instances**(class,description)
description ::= id-of-a-dataclass-instance
| list-of-attribute-specifications | **all**
attribute-specification ::= [ attribute, bound-description ]
| [ attribute, **range**(lower-bound, upper-bound) ]

The boolean function *Describes?(B,I)* is assumed to determine whether the bound-description B covers the instance I, and the function *Prune(SB)* eliminates from the set SB of bound descriptions those which are subsummed by others.

*(ii) Other terms: minclass(e)* returns the set of lowest classes in the IsA hierarchy to which e belongs, if e is an entity; if e is a collection of classes, then minclass(e) is the set of its least upper bounds in the IsA partial-order. In addition, *attribute*(p,C1,C2) states that for class C1, property p is defined to have range C2. In what follows, variable names ending in Seq will be sequences of values, and the notation Seq[++n] means that n is first incremented and then used as an index to refer to the nth value in the sequence. Finally, the global variable CO will contain the constraint whose violations are being generalized.

*(iii) The algorithms:*

***function*** DESCR (ExSeq:list of exceptional instances)
returns a set of bound-descriptions
if scalars(ExSeq) then return(**range**(min(ExSeq),max(ExSeq)))
someEx = ExSeq[1]
B = { **instances**(c,someEx) | c ∈ minclass(someEx) }
for each exception ex ∈ ExSeq where ex≠someEx do
  B1 = {}
  for each bound-description b ∈ B do
    B1 = B1 ∪ GENERALIZE(b,ex,1)
  B = Prune(B1)
return(Prune({ CROP(b,1) | b∈B }))

```
function GENERALIZE(b:bound-description,I:instance,level:int)
        returns a set of bound-descriptions
suppose b ≡ instances(c,d)
if Describes?(b,I) then return({b})
B1 = {}
if ¬isin(I,c) then
  C = minclass({c} ∪ minclass(I))
  for each class k ∈ C do
    B1 = B1 ∪
        GENERALIZE(instances(k,APPLICABLE(d,k)),I,level)
else if d is an id-of-a-dataclass-instance then
  for each bound-description bd ∈ SPECIFY(d,c,level) do
    B1 = B1 ∪ GENERALIZE(bd,I,level)
else /* isin(I,c) ∧ d is a list-of-attribute-specifications */
  n = 0
  for each attribute-specification as ∈ d do
    if Describes?(instances(c,as),I) then deSeq[++n] = { as }
    else case on the form of as do
      [attr, range(lower,upper)]:
        lower = min(lower, I.attr)
        upper = max(upper, I.attr)
        deSeq[++n] = { [attr, range(lower,upper)] }
      [attr, instances(c2,d2)]:
        deSeq[++n] = { [attr, b] | b ∈
          Generalize(instances(c2,d2),I.attr,level+1)) }
  B1 = { instances(c,las) | las ∈
          cartesian product of the n sets in deSeq }
return(Prune(B1))

function APPLICABLE(D:description, k:class)
        returns description
/* eliminates from D those attrib-specifications
   whose attributes are not defined for k */
if D is not a list-of-attribute-specifications then return(D)
n = 0
for each attribute-specification as ∈ D do
  suppose as ≡ [attr, spec]
  if attribute(attr,k,c) ∧ c≠None then asSeq[++n] = as
if n=0 then return(all) else return(asSeq)

function SPECIFY(I:instance, d:class, level:integer)
        returns a set of bound-descriptions
/* introduces a list of attribute-specifications
   for the dataclass instance I of class d */
n = 0
if level≤nestingThreshold then
  for each attribute p of class d do
    suppose attribute(p,d,c)
    if level≠1 ∨ constraint CO is not
      the attribute range constraint on p then
        if c is a dataclass then
          SC = minclass(I.p)
          if SC={} then SC = { c }
          deSeq[++n] = {[p,instances(c1,I.p)] | c1∈SC}
        else deSeq[++n] = { [p, range(I.p,I.p)] }
if n=0 then return({ instances(d,all) })
return({ instances(d,asSeq) | asSeq ∈
          cartesian product of the n sets in deSeq })

function CROP(b:bound-description, level:integer)
        returns a bound-description
/* removes attribute specifications
   which are redundant or not relevant */
suppose b ≡ instances(c,d)
if d is not a list-of-attribute-specifications then return(b)
n = 0
for each attribute-specification as ∈ d do
  relev = RELEVANT?(as,c,level)
  case on the type of as do
  [attr, range(lb,ub)]:
    if attribute(attr,c,range(lb,ub))
      then /* redundant so omit */ nil
    else
      if relev and fewer than
            ProportionThreshold[level] % of instances
            of c have values for attr in the range(lb,ub)
        then asSeq[++n] = as
        else if ¬CAN_DROP(attr) then  asSeq[++n] = as

  [attr, instances(c3,d3)]:
    b2 = CROP(instances(c3,d3), level+1)
    suppose b2 ≡ instances(c3,d4)
       and attribute(attr,c,c2)
    if d4=all ∧ c2=c3
      then /* redundant so omit */ nil
    else if d4 is an id-of-a-dataclass-instance
      then asSeq[++n] = as
    else if relev and fewer than
            ProportionThreshold[level] % of instances
            of c have values for attr described by b2
      then asSeq[++n] = as
      else if ¬CAN_DROP(attr) then  asSeq[++n] = as

if n=0 then return(instances(c,all))
else return(instances(c,asSeq))

function RELEVANT?(AS:attr-spec, C:class, level:integer)
        returns boolean
suppose AS ≡ [A, S]
x1 = 1/(1+IsA-distance(C,Class-where-Introduced(A)))
if A occcurs syntactically in constraint CO
  then x2 = 1 else x2 = 0
if (wt1*x1 + w2*x2) < RelevanceThreshold[level]
  then return(false) else return(true)

function CAN_DROP(Attr:attribute) returns boolean
/* global check to see if unacceptably many negative
   examples (detected errors) would have been allowed by
   the modified integrity constraint, if the description
   of exceptions was modified by dropping the specification
   of this attribute. */
```