Language Features for Flexible Handling of Exceptions in Information Systems

Alexander Borgida[1]
Department of Computer Science
Rutgers University

Abstract

We present an exception handling facility suitable for languages used to implement database-intensive Information Systems. Such a mechanism facilitates the development and maintenance of more flexible software systems by supporting the abstraction of details concerning special or abnormal occurrences. We consider the type constraints imposed by the schema as well as various semantic integrity assertions to be normalcy conditions, and the key contribution of this work is to allow *exceptions* to these constraints to *persist*. To achieve this, we propose solutions to a range of problems, including sharing and computing with exceptional information, exception handling by users, the logic of constraints with exceptions, and implementation issues. We also illustrate the use of exception handling in dealing with null values, estimates, and measurements.

Keywords and phrases: semantic integrity, violations of type constraints, exception handling, accommodating exceptions, conceptual models
CR categories: D.2.5 (Error handling and recovery), H.2.0 (Database integrity), H.2.1 (Data models), H.2.3 (Database DDL/DML languages),

This work has been supported by the National Science Foundation under Grant No. MCS-82-10193.

[1]Author's address: Department of Computer Science, Rutgers University, New Brunswick, NJ 08903.

1. INTRODUCTION AND MOTIVATION

Computerized information systems (ISs henceforth) for applications such as personnel records, credit and billing, reservations, etc. manipulate large amounts of data and are designed with the goal of assisting humans in their tasks in the business world. The primary goal for such systems has in the past been the efficient storage and retrieval of information under normal circumstances. One of the distinguishing characteristics of intelligent human behavior in the natural world is

FLEXIBILITY: the ability to deal with unusual, atypical or unexpected occurrences.

In this regard however, practically all current commercially developed software is the exact antithesis of the human system that it replaces or assists: programs are extremely rigid and intolerant of deviations from the norm set out by their designer. For example, in relational database management systems it is not possible to store a tuple that does not conform to the schema (e.g., if it has additional attributes or values outside the domain specified for an attribute); similarly, it is not possible to allow violations of integrity constraints to persist without turning off completely the checking of those constraints. As a result, ISs have been unable to fully support the activity of their users. Many of the "horror stories" in the popular press about the misadventures of citizens dealing with computerized organizations are also directly attributable to the rigidity of the software in place.

The goal of this paper is to present language features that make it easier to develop ISs having the desired degree of flexibility, as this term was defined earlier. One reason why it is difficult to write flexible software is that the designer must constantly take detours from writing code that deals with normal cases, in order to account for the multitude of unusual circumstances. This is an unsystematic process that leads to incomplete and complicated code. A long-recognized principle of software engineering and programming language design is that languages must support the notion of *abstraction*: the ability to consider only some of the details of an entire problem while others, presumably less important, are suppressed until some later step in a refinement process. We believe that it is useful to be able to ignore at first those circumstances which are special, rare or unusual, so that the designer can concentrate on the usual or normal state of affairs. Special cases should then be considered later, and code to deal with them should be added in the form of annotations. This abstraction principle, which we shall call *special case deferment*, is important not only during the design process but also during software maintenance: a program is more understandable if one can first get an impression of what it does normally, without constantly being distracted by special cases.

Deferment of special cases, which has not been explicitly recognized as an abstraction principle until now, is particularly important for the development of application programs that utilize data bases, since studies have shown that a significant portion of such programs is devoted to error or special case checking and handling. The central role of exception specification has also been recognized in Office Automation -- see for example the work of Kunin [18]. Like aggregation and generalization [25], special case deferment is therefore an abstraction principle particularly suitable for the task of IS design. In addition, we will argue that not all special cases can be anticipated at the time when IS programs are written, and hence we will require a mechanism that will also allow users of ISs to deal with these special cases *at run-time*.

It turns out that the abstraction of normal cases is supported in some programming languages by the *exception handling* mechanism, initially designed to deal with errors arising from built-in operations (e.g., division by 0). Our work then extends the pioneering explorations of exception handling in programming languages for IS design, initiated by Wasserman [29] and the Taxis project [22].

One of the reasons databases are important to an organization is because they allow standardization of the information form and provide for centralized control of the quality of information. Data Base Management Systems (DBMS) help achieve these goals by allowing constraints to be imposed through the *schema* and various *integrity constraints* (ICs) of the database. Constraints whose enforcement is not supported by the DBMS - and many DBMS are deficient in this regard - are checked by the IS application programs used to modify the database. All these constraints are intended to maintain the database free of errors arising through carelessness or lack of knowledge; some are also used by the DBMS and compilers to optimize the storage, access and processing of data. The key contribution of our research is to view these constraints as just normalcy conditions - ones that may occasionally conflict with the actual state of the world. For this reason, we need the ability to permit violations of constraints to persist, i.e., to *accommodate exceptional facts and occurrences*. Since until now one could only reject updates that led to inconsistent database states, our work is a natural development in the area of *failure actions for semantic integrity constraints* in data bases; it thus complements previous research on the specification and enforcement of such constraints [14], [10], [27], [9], [6], [2], [17]. Allowing exceptions to persist raises a host of issues, which will be discussed in further detail in the next section, and which are a chief concern of this paper. Without going into details, we note here that we will rely on the general exception handling mechanism to accommodate violations of constraints; the concept of "handling exceptions" thus provides a unifying framework for our research. We present next more detailed arguments and examples illustrating the need for occasionally suspending the constraints stated for a database. We also classify the problems that will be encountered if such suspensions occur.

1.1. Motivating examples and problems

An IS imposes constraints on the form and nature of the information stored, as well as the access and modification of this information. For example, if a utility company used a relational DBMS, the following constraints would reasonably be included in its design:

- A customer has a Social Security number that uniquely identifies him.
- A customer has one address, and addresses have the standard form of street, city, state and zip code, with states and zip codes having a list of legal values.

- Each customer is associated with his most recent power meter reading, and this value cannot decrease when updated.
- The *amountDue* of a residential customer is a dollar amount between \$0 and \$200, and it is computed from successive meter readings.

Unfortunately, IS designers are faced with a significant problem in stating and using constraints: ISs are usually used to store information about the real world, and philosophically speaking this world is populated by *natural kinds* [24] -- concepts like persons, readings, hirings. In contrast to mathematical concepts such as "triangle" or "sequence", natural kinds do not have precise definitions, and assertions about such concepts are either too general, to the point of being vacuous, or can only be said to hold "typically". Unfortunately, the current notions of schema and integrity constraints presuppose the existence of precise definitions and constraints for natural kinds. The following examples illustrate situations where the previously stated constraints might be violated for the hypothetical utility company; at the same time, they indicate the difficulty of generalizing or eliminating the constraints in order to avoid contradictions.

- When a mechanical error (e.g., two meters had been cross-connected between two apartments) is rectified, the latest reading may in fact be lower than the previous one.
- Newly arrived immigrants or visitors usually do not have a Social Security number at the time they start renting an apartment. If the presence of a few exceptions eliminates the use of this number as a unique identifier ("key"), then we need to introduce an additional, artificial key, which is computationally expensive (assuming Social Security numbers are still needed), as well as a burden on users.
- A customer may leave for vacations and may wish to have bills forwarded to his winter residence in France. If we were to allow for this in the design, we would have to permit arbitrarily complex addresses from around the world. Since describing all possible correct addresses in the world is an impossible task, we would have to forego checks like the ones for valid state names and zip codes.
- A customer may request to have bills forwarded to *two or more* addresses because he is uncertain where he will be in the next few months.
- A few persons may have monthly bills exceeding \$200. If we were to increase the limit to \$1000 or even \$10000, we would lose the ability to catch common data entry errors such as adding extra digits or dropping decimal points, which was the main purpose of the constraint.
- Suppose that a customer sends a payment that is larger than the current *amountDue*, and the company wishes to credit it towards next month’s bill. One way would be to record the surplus as a negative *amountDue*, which violates our constraint on this attribute. Another would be to set the amount due to \$0, but record elsewhere the surplus, and ensure that next month’s bill is appropriately reduced; the question is where such a surplus should appear in the database.
- In a rare case, the company may obtain non-cash reimbursement, such as some real estate, as payment for a large delinquent account. The company wishes to keep it in the hope that its value will increase in the future, but the current value of the asset needs to be used for tax purposes at the end of each year. Such conversions and special cases are notoriously difficult to deal with in data processing (and accounting).

Kent [16], in his critique of record-based DBMS, cites additional evidence of the fact that the world is sufficiently "inhomogeneous" to make strict constraints unworkable. There is thus evidence that almost any rule will have exceptions, and that there is no reasonable way to anticipate exceptions and generalize rules in order to avoid contradictions. The alternative of excluding contradictory information from the database is also unacceptable, since it leads to inaccurate, incomplete and out of date information; it also often results in the maintenance of parallel manual files that are much more volatile, no longer easily shared, and outside the control of management. As a result we are led to consider the possibility of having constraints *coexist* with information which conflicts with them. In particular, we have seen evidence that

- dynamic integrity constraints on the ways in which the database is updated, as well as constraints on the inter-relationship of values may need to be suspended;
- attributes with scalar values, such as integers or strings, may be assigned values outside specified subranges;
- attributes may be assigned values that are of completely different basic types (e.g., real estate as opposed to number);
- some objects may require multiple attribute values where single values were expected (e.g., addresses), or they may require additional, unexpected attributes (e.g., *overPayment* for a customer, or *country* for an address).

Once we accept the need to allow deviations from the norm set out during IS design, we must consider a number of problems, which fall in the following categories:

Semantics of computations: Exceptional data will need to be treated with circumspection in future computations, since its semantics may be different. For example, although all US addresses have a zip code attribute, a foreign address may not have one. Similarly, when a numeric attribute, such as temperature, is assigned a non-numeric value, such as "normal", then it cannot be used in arithmetic computations.

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 that normally holds an integer representing US dollar values now has Swiss francs in it.

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. In a shared database, accountability for allowing rules to be violated must also be established. It is thus necessary to keep a log of the circumstances surrounding each exceptional occurrence and the person taking responsibility for it.

Validation of future updates: 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 constraint will not be able to detect errors in later updates since it is unable to differentiate between an erroneous new update and a false alarm due to the old exception.

Efficiency of data storage and manipulation: Although we are willing to spend additional time processing exceptional data, we don’t want the presence of exceptions to degrade the efficiency of handling normal data. For example, information about the type of values stored in an attribute is used by the DBMS to reduce space and retrieval costs (e.g., fixed-length record-storage schemes), and by compilers to optimize programs (e.g., remove run-time checks). This must be made to coexist with exceptions to type constraints.

1.2. Overview

In Section 2 we present a language for implementing Information Systems which shall be used for examples, and in Section 3 we introduce our exception handling mechanism for this language. The features of the exception handling mechanism that allow violations of type and integrity constraints to persist are presented in Sections 4, 5 and 6. In Section 7, we investigate the First Order Logic of constraints with exceptions, thereby providing a formal foundation to the concept fo "exception", and in Section 8, we make some suggestions concerning implementation problems raised by our approach. In Section 9, we show that our mechanism has wider applicability by addressing several problems in databases such as various null values, estimates, measurements, multi-valued or new attributes, and exceptional instances of classes. We conclude by evaluating our mechanism and the key ideas underlying it.

2. THE FRAMEWORK OF CONCEPTUAL MODELING LANGUAGES

As the specific context of our research we will use a programming language that allows the definition of ISs at the conceptual level [4]. This language is an adaptation of the language Taxis [22], with some influences from Galileo [1] and ADAPLEX [26]. We present here only those aspects needed to provide a context for exception handling, and we refrain from using esoteric features in order to make the results of our investigations more widely applicable.

We view an IS as a *model* of the world about which it contains information. The world is seen as populated by *objects*, which are inter-related through *properties* (or *attributes*). The designer controls the quality of information stored in an IS by describing generic *classes of objects* and insisting that every specific object belong to at least one class.[2] The information about objects can be queried and modified either by ad-hoc queries and commands, or by *transactions*, which can be thought of as models of activities in the world [5]. In an *interactive IS*, such transactions are directly invoked by end-users, who are assumed to be present during their execution.

2.1. Defining classes of entities

Certain classes of objects are used to represent traditional data items in databases. In addition to such primitive classes as INTEGER and STRING, programmers may define subranges

```
    $VALUE := 0.00 .. 100000.00
```

and enumerations

```
    RATINGS := {low, high}.
```

Other classes of objects are described in terms of the properties attributable to their instances together

[2]Languages supporting these concepts, as well as taxonomies of classes, will be called *object-oriented* languages.

with constraints on the possible values of these properties, as in the definition of the classes ADDRESS and CUSTOMER in Figure[3] 1.

```
entity class ADDRESS with
    attributes
            apt#: 1..1000
            house#: 1..10000
            street: STRING(25)
            city: STRING(25)
            state: {AL,..., WY}
            zip: 00000..99999
    end
entity class CUSTOMER with
    attributes
            name: STRING(25)
            address: ADDRESS
            meter: METER
            amountDue: 0.00  .. 100000.00
            creditLimit: 50.00 .. 100000.00
            creditRating: RATINGS
    end
```

Figure 1: The ADDRESS and CUSTOMER classes.

Objects corresponding to individuals in the world, such as customers, are called entities. The above definition allows instances of the class CUSTOMER to have attributes *name, address,* etc. and constrains the values of these attributes to being members of the classes specified to the right of the colon. A *name* must therefore be a string, an *address* must be an object in ADDRESS, and amounts due are values up to 100000.00. Entity classes such as CUSTOMER have instances explicitly specified through update operations.

Classes can be organized in subclass hierarchies (e.g., INDUSTRIAL_CUSTOMER is a subclass of CUSTOMER), where at any time all instances of a class must also be instances of its superclasses. The definition of a subclass need not duplicate the definition of its superclasses: it is enough to present additional property definitions or refinements. This abbreviatory device is known as *inheritance* of attributes and constraints. For example, according to the example in Figure 2, residential customers are customers who have in addition Social Security numbers, and have their monthly payments limited to $200, rather than the more general $100,000.

```
entity class RESIDENTIAL_CUSTOMER   IsA   CUSTOMER with
    attributes
        socSec#: STRING(9)
        amountDue: 0.00 .. 200.00
        . . .
    end

entity class INDUSTRIAL_CUSTOMER   IsA   CUSTOMER with
```

[3]Figures referred to in multiple places appear in Appendix A.

```
attributes
    placeOfIncorporation: STATE
    . . .
end
```

Figure 2: Two partially-specified subclasses of CUSTOMERs

There are several pre-defined "maximal" classes in the hierarchy, including ANY_ENTITY, which has all entities as instances, and ANY_PROPERTY, which has all property identifiers as instances.

We have seen already that class definitions provide *type constraints* on their instances by specifying which attributes are applicable to them and by restricting, through *attribute type constraints*, the possible range of values for each attribute. More complex constraints on entities can be stated as *logical constraints* expressed in a variant of First Order Logic, whose exact details are not relevant here. This is accomplished through properties whose values are assertions, which are supposed to evaluate to true. These assertions involve the object, its properties, and possibly other objects introduced through quantification. All constraints are classified as **initial, final** or **always**, determining the time when they should be checked: initial constraints are verified at the time the object is added to that class, final constraints when it is removed, and **always** constraints must hold after every update. For example, according to the example in Figure 3, residential customers are required to have no balance due when they are created and destroyed, the outstanding balance cannot exceed the credit limit, and the Social Security number must always be known and distinct for each customer.

```
entity class RESIDENTIAL_CUSTOMER IsA CUSTOMER with
  attributes
          ...
  initially
          startClean!: (amountDue = 0)
  finally
          noDebt!: (amountDue = 0)
  always
          underLimit!: (amountDue ≤ creditLimit)
          knownSS#!: (socSec# ¬= nil)
          keySS#!: for every z in CUSTOMER (socSec# = z.socSec# ⇒ this=z)
```

Figure 3: Constraints attached to RESIDENTIAL_CUSTOMER class[4]

The preceding technique allows us to express all integrity constraints of the form $(\forall x \in C)\, \Phi(x)$ by attaching the assertion Φ(**this**) to the objects in class C. Other constraints, such as those starting with existential quantifiers or aggregate operators such as SUM or AVERAGE, can be made assertion properties of classes themselves, rather than of their instances. For example, the constraint that the sum of all amounts due by instances of customers (instances of class CUSTOMER) cannot exceed $500,000 would become the *debtCeiling* property of CUSTOMER, where CUSTOMER is considered to be an object

[4]The reserved keyword **this** refers to the object whose property is being evaluated, and the notation x.p is used to refer to the value of property p for the object in variable x; we abbreviate **this**.p as p

itself.

Expressing semantic integrity constraints as assertion-valued properties of entities brings out their similarity to attribute type constraints; it is a natural extension to object-oriented languages, and this is the principal new contribution in this section.

2.2. Data manipulation and transactions.

In this data model, information is recorded by the presence or absence of objects as members of classes, and by the value of their properties. This information can be updated by primitive operators such as:

```
modify socSec# of y to 422344117  (*  y.socSec# ← 422344117 for short *)

create x in CUSTOMER with  name:..., ... ;
        (* a new, distinct object is created with the specified property
          values; it is made an instance of class CUSTOMER; the variable
          x appears optionally  and is assigned the newly created object
          as value; *)


delete y; (* destroy an object; in order to prevent dangling references, this
           is only permitted if the object which is the value of  y is not
           the property value of some other object *)
```

Information in the database can be obtained through the use of retrieval operators such as

```
    evaluate amountDue  of y  (* abbreviated as  y.amountDue  *)

    get y from CUSTOMER with socSec#=789789789 such that <boolean expr.>;
        (* abbreviated as  y ← CUSTOMER⟨socSec#=789789789⟩ such that ...
          this operation assigns the variable y some instance of CUSTOMER with
          the desired property values, if one exists *)


    y isIn? RESIDENTIAL_CUSTOMER  (*  boolean-valued membership test *)
```

For uniformity, a transaction definition is also viewed in Taxis as a class, whose instances are particular invocations of that transaction. The transaction corresponding to the reading of a meter is illustrated in Figure 4. Note that the parameters and local variables of the transaction are considered to be properties of transaction objects, and are subject to type constraints. Transactions also have logical constraints on their proper instantiation (invocation), expressed as assertions. These constraints are classified as either **preconditions** or **postconditions** depending on whether they are expected to hold at the time the transaction is initiated or terminated, respectively. Such conditions are needed for stating transition constraints (e.g., meter readings cannot decrease), but can also be used to prevent or detect violations of other integrity constraints, which could have been stated as logical constraints on entity classes. For example, aggregate constraints can be checked as postconditions. The decision to build in such checks into transactions is taken on grounds of efficiency, and must be weighed against the possibility that some programmer may forget to add a check of this constraint to his transaction.

```
transaction class READ_METER( mtr, newReading, newDate)  with
   variables
           mtr: METER
           newReading: INTEGER
           newDate: DATE
           ...
   preconditions
           consumedMore?: (newReading ≥ mtr.lastReading)
           later?: (newDate ≥ mtr.lastReadingDate)
   actions
           mtr.lastReading ←  newReading;
           mtr.lastReadingDate ←  newDate;
   end
```

Figure 4: The READ_METER transaction.

In addition to the primitive operators for manipulating objects, the language supports compound statements such as conditionals (IF/THEN, IF/THEN/ELSE) and loops (WHILE, FOR <var> IN <class> SUCH THAT... DO).

The concept of a transaction corresponds to that of a procedure in ordinary programming languages - both "function" and "subroutine" transactions are available. The principal difference is that a transaction is also a unit of *recovery* in case of failure (including hardware and concurrency problems). As with ordinary procedures, it is desireable to to allow nested transaction calls and definitions. For this reason, we adopt the view that each nested transaction invocation sets up an intermediate "save point", to which one can back-up if necessary (a crash) or desired (UNDO command); however, the effects of a nested transaction invocation are not committed until the *top-level* transaction is completed. This model resembles that of the System R recovery manager [12], and it provides, at no extra cost, the concept of "save point", which we use both in defining our exception-handling mechanism and in providing programmers with an explicit **undo** statement.

For complete uniformity, we view the primitive operators **modify, evaluate, create, get**, etc. as pre-defined transactions $MODIFY, $EVALUATE, $CREATE, $GET, etc. Among others, these operators enforce the constraints specified by entity class definitions. For example, we can view the **modify** operator as having the schematic structure in Figure 5.

```
transaction class $MODIFY(obj, prop, newVal)  with
   variables
           obj: ANY_OBJECT
           prop: ANY_PROPERTY
           newVal: ANY
   preconditions
           exists?: (obj ¬= nil)
           propertyApplicable?:  there is a class  to which obj belongs and
                which has prop defined for it;
           valueInRange?:  newVal belongs to the class specified for prop by
               all the classes to which obj belongs;
           others?:  all other semantic constraints stated on classes continue
           to hold;
```

```
actions
          store newVal as the value of obj.prop;
end
```

Figure 5: The primitive operator $MODIFY as a transaction.

3. AN EXCEPTION HANDLING MECHANISM

3.1. Overview

We will motivate and describe our exception handling mechanism in several passes, each of which will cover additional details. To begin with, we introduce the programming language terminology for exception handling using concepts in [20], and outline the basic mechanism for exception handling, which in many ways lies within the framework originally established by Goodenough [11] (viz. his "signal" exceptions).[5]

In any large program system, one must face the problem that many operators are not defined for all possible inputs (e.g., division by 0, pushing on a full stack). It is often desirable to react to such "error situations" in a context-dependent manner: when underflow occurs in an arithmetic computation, it is acceptable to use 0 as a result in some cases; in others, the entire computation must be abandoned. Most modern programming languages support the notion of *exception* and *exception handling* for the purpose of specifying what is to happen when such special cases arise, and generalize this mechanism to allow user-defined "errors". The following description addresses briefly the issues of exceptional situation detection, signaling, diagnosis and recovery.

In general, transactions (primitive or user-defined) invoke others to perform some task. Suppose that one transaction -- the *caller* P -- has textually embedded an *invocation* of another transaction S. When the invocation of S is executed, the result is an *activation* of S. This activation may encounter an abnormal situation, and as a result *signal an exception* E. This invocation is said to *raise* the exception E, and S is called a *signaller*. The program text intended to be executed when that particular exception is raised is called the *handler* H, and an obvious place to specify it is at the place where the caller invoked the signaller:

```
P
  ...
  S;  [E  =>  H]
  ...
```

Note that different handlers for exception E may be specified on distinct invocations of S in P, and that a transaction S may on different occasions signal different exceptions. If a handler for this exception is not specified on the invocation of the *initial signaller*, the exception is *propagated*, so that the caller P is

[5]Significant ideas on exception handling in programming languages have also been presented in [29], [19], [15], [8], [30]

interrupted and it raises E itself, thus becoming a signaller. This propagation up the invocation chain continues until either some caller provides a handler for E on the invocation of its signaller or the top-most transaction invocation is reached. In the latter case, the user can provide a handler or some default handler is invoked. Before an exception handler is located, there may therefore be an initial signaller, zero or more intermediate signaller, and a final signaller; heretofore, *the* signaller will refer to the final signaller.

As customary, handlers can be specified on syntactic units (statements or expressions) which enclose the invocation of the signaller. The easiest way to present this is to view compound statements such as conditionals, loops and begin-blocks as operator invocations (as in LISP for example). Thus, the invocation of transaction T defined by the fragment

```
transaction class T(x) with
        . . .
        if (y.age > 60)
           then begin
                y.amountDue ← 0.8 * y.amountDue;
                ...
                end;
        . . .
  end;
```

gives rise at some point to an invocation chain that includes T, $IF, $BEGIN, $MODIFY, $FLOAT_MULTIPLY, and $EVALUATE. Therefore, a handler for some exception E raised by $EVALUATE could be specified on the invocation of $MODIFY:

```
y.amountDue ← 0.8 * y.amountDue;  [E => V]
```

Once a handler for some exception is determined, it is executed and then flow of control may continue in one of two ways:

- Signaller termination: The activation of the signaller (and all its embedded activations) is terminated at the point where it raised the exception, and execution continues at the point right after the invocation of the signaller. In this case, the handler can be seen as replacing the signaller.

- Signaller resumption: The signalling activation(s) may be be continued at the point where the exception was initially raised.

The following summary outlines how we will apply the above framework to the problems of database constraints discussed earlier: We will consider violations of logical and type constraints as exceptions, and the exception handling mechanism will be used to specify failure actions. The termination mechanism will be used to specify corrective actions when errors arise (see Section 3.2); the resumption mechanism will be used to suspend constraints in order to accommodate exceptions (see Section 4). More interestingly, we will also use the exception handling mechanism to deal with the problems of sharing and computing with exceptional information: exceptional facts will be marked and exceptions will be raised whenever such facts are accessed (see Section 5). We will describe additional techniques necessary to deal with the suspension of semantic integrity constraints in Section 6.

3.2. Violation signalling and handling.

An exceptional situation will be said to arise when some constraint is violated. Therefore, an exception is initially raised by a transaction when one of its pre or postconditions evaluates to false (as in Taxis), or by a primitive operator of the language when either a built-in condition or assertion specified on some entity class turns out to be false (see description of $MODIFY in Figure 5, for example). In view of this definition, we call such exceptions *violations*.

The occurrence of a violation is signalled by the creation of an object, distinct from other objects. As all other objects in the language, the violation object must belong to some class, which in this case will be called ANY_VIOLATION (see Figure 6). Violation objects have properties identifying the constraint being violated (*pr* and *cl*), and allowing information to be passed out from the signaller. Observe that all constraints stated by the programmer are uniquely designated by two identifiers: those of a class and of a property. For example, the constraint that CUSTOMERs must have numbers between 0.00 and 100000.00 as *amountDue* is identified by (CUSTOMER, *amountDue*), while the constraint that successive meter readings must increase is identified by (READ_METER,*consumedMore?*).

```
exception class ANY_VIOLATION with
  attributes
    cl: ANY_CLASS
    pr: ANY_PROPERTY
    op: ANY_TRANSACTION
  end
```

Figure 6: Definition of the class of violations

For this reason, when a violation object is created, *cl* and *pr* are assigned so that they identify the assertion that failed. The property *op* provides access to the specific circumstances of this violation occurrence. In the case of constraint failures for transactions, the value of *op* is the object corresponding to the transaction activation that detected the violation; in the case of entity class constraints, *op* is the predefined operator ($MODIFY or $CREATE) that found the violation. In either case, information about the parameters of the operation is accessible to the programmer as properties of the object corresponding to the operator invocation.

In order to allow easier identification, and in order to allow the same signal to be raised in distinct circumstances (under the assumption that all these violations will have to be handled the same way), a programmer can define new subclasses of ANY_VIOLATION in the same way as defining new subclasses of entity classes. In Figure 7, we illustrate the definition and use of one such violation class, METER_READ_VLN.

```
exception class  METER_READ_VLN   IsA  ANY_VIOLATION;

transaction class READ_METER( mtr, newReading, newDate)  with
        . . .
  preconditions
     consumedMore?: (newReading ≥ mtr.lastReading) else METER_READ_VLN;
```

```
        later?: (newDate ≥ mtr.lastReadingDate) else METER_READ_VLN;
           . . .
    end
```

Figure 7: Associating new violations with assertions

Programmer-defined violations can be attached through **else** clauses to both logical constraints and attribute type constraints on entity and transaction classes. In addition, there are a number of predefined violation classes, such as DIVISION_BY_ZERO, ATTRIBUTE_TYPE_VIOLATION and INAPPLICABLE_PROPERTY. Note that although only one violation object is created when an exception occurs, the taxonomy of violation classes gives the effect of multiple violations since this object can be in several classes.

We view a handler as a transaction, which may access a special variable **thisViolation**. The value of this variable is the violation object that led to the invocation of the handler. For convenience, we often omit the definition of the handler transaction and give only its actions.

The association between a violation and its handler is specified syntactically as

```
S [V => H]
```

which indicates that if this activation of S signals or propagates violation V, then handler H should be executed. For example, the program below indicates that if an instance of METER_READ_VLN is raised by READ_METER, then a corresponding error message is to be printed:

```
READ_METER(x, val ,$Today)   [ METER_READ_VLN  =>
                               write('Constraint',thisViolation.pr,
                                       'is violated for READ_METER');  ]
```

The variables visible to the invocation of H are those originally visible to the invocation of S. Furthermore, all updates performed by the transactions activated by the invocation of S are considered undone. The intention of these conditions is to make the invocation of the handler appear as a replacement of the invocation of the signaller. The execution of the handler is considered to be terminated when its last statement is executed or when a **return** statement is encountered[6]. When a handler terminates in this way, the violation object that led to its invocation is destroyed. If the activation of a handler itself raises a violation, violations are handled in a last-in first-out manner.

The specification V of violations that are handled by H can in fact be a predicate describing a violation object using the syntax of conditions for the **get** operation. In the following example, the handler will be invoked any time the *underLimit!* assertion fails for a residential customer:

```
y.amountDue ←  150;
      [ ANY_VIOLATION⟨cl=RESIDENTIAL_CUSTOMER, pr=underLimit!⟩   =>
        y.amountDue ← 100 ]
```

Note that in this case it was $MODIFY that raised the violation concerning the

[6]If S is a function, then H must return a value, which is of the same type as expected by the call to S.

RESIDENTIAL_CUSTOMER integrity constraint *underLimit!*.

As mentioned earlier, if an applicable handler for a violation is not found on the invocation of the signaller, then the violation is propagated up. Thus, in the next example the entire IF-statement would be abandoned after a violation of the range constraint for *amountDue* of customer y:

```
if (y.age > 60)
   then  begin
        y.amountDue ← y.amountDue - 100.00;
        gifts ← gifts + 100.00;
        end;   [ANY_VIOLATION⟨cl=CUSTOMER,pr=amountDue⟩  => doNothing ]
```

If more than one handler for a violation object is found on some invocation, then one of them is chosen nondeterministically.

To specify some standard behavior in the absence of explicit handlers in a particular program, a programmer can indicate a default handler for a violation by attaching it to the definition of the corresponding violation class through the special property *defaultHandler*. The default handler of the least violation class to which the violation object belongs is invoked at the top level.

4. RESUMING AFTER VIOLATIONS OF ASSERTIONS.

In order to accommodate exceptional situations such as the ones discussed in Section 1.1, we provide the **resume** statement for use in handlers. As the name indicates, execution of this statement in a handler results in the resumption of the execution of the initial signalling transaction -- namely, **thisViolation.***op*. Thus, the following code has the effect of suspending constraints *consumedMore?* and *later?* on this invocation of READ_METER

```
        READ_METER(x,4000,OldDate)   [METER_READ_VLN  => resume]
```

When an operation is resumed, it continues immediately after the place where the violation was detected. After a constraint was suspended by resuming its violation, the checking of another constraint may lead to another violation, which must also be handled. Therefore violations of multiple integrity constraints are raised and handled serially.

The following model explains the possibility of a handler resuming the signaller. We emphasize that this is merely a "mental model", which does not preclude alternate implementation techniques. When a procedure is activated, a *frame* is created to hold its local variables and control information.[7] Normally, these frames are kept on a stack and every frame has an associated *link*, which indicates the frame of the invoker, and which provides access to global variables. In the case of an exception, we imagine that our stack branches out to become a tree of merged stacks (see diagram below): In this structure, the frames of all signallers, including the initial one, continue to exist alongside that of the handler, which is linked to the frame of the caller that provided the handler. The diagram illustrates the case when P contains the

[7]This is a very simplified version of a model of process activation described in [3].

statement *Q [E => H]* , and S signalled E inside this activation of Q.

```
-------
|  P  |
-------   \
   |        \
-------       \ -------
|  Q  |         |  H  |
-------         -------
   |

   |
-------
|  S  |
-------
```

If the handler H terminates by **return**-ing, then the frames for S,..., Q are abandoned; if the handler terminates by **resume**-ing, then the frame for H is discarded. In fact, we consider it illegal for a handler that resumes to have modified anything but its local variables. Otherwise, the handler could change properties of database entities that had been earlier verified by some transaction precondition; this would allow the circumvention of constraints without violations having been raised.

The following more complex example illustrates the possibility of a handler either resuming or terminating the signaller based on external considerations:

```
READ_METER(x, value, someDate);
                [ANY_VIOLATION⟨cl=READ_METER,pr=consumedMore?⟩ =>
                ARE_YOU_SURE]
```

where ARE_YOU_SURE is defined as

```
transaction class ARE_YOU_SURE  IsA  VIOLATION_HANDLER with
    variables
        answ: STRING(1);
    actions
       write('The current reading of this meter is', x.lastReading);
       write('Do you wish to lower this to', value);
       write('<Answer Y or N>');
       read(answ);
       if answ='Y'
          then  resume
          else  begin
               write('Please type in correct value:');
               read(value);
               READ_METER(x,value,$ToDay)
               end;
       end; (* of ARE_YOU_SURE*)
```

In this case the user is asked to decide whether the original program was invoked with the correct arguments or not.

The **resume** command allows exceptions to transaction pre and postconditions to persist. To make some condition of a user-defined transaction "inviolable", the programer can add a second explicit test of this condition, and then **undo** the transaction if it is found to be false. More problematic is resumption following the violation of some constraint raised by a primitive operator. We will devote the next section to the desired behavior of $MODIFY, $CREATE and $EVALUATE after resumption. As far as the other primitive operators are concerned, including arithmetic and logical operators, we extend the language with the special constant **unv**, which stands for "unusual value" or, better still, "un-value" or "uninterpreted value". Most partial functions are extended to take **unv** as their value outside their current domain, and any function with **unv** as an argument has **unv** as value; the sole exception is the predicate UNVALUE?, which tests if a value is **unv**. Therefore, if one resumes the division of 2 by 0 after violation DIVISION_BY_ZERO was signalled, the result will be **unv**; similarly, (4<**unv**) yields **unv**. An exception to this is resuming a conditional statement after the condition did not evaluate to **true** or **false**: this seems to have no reasonable semantics, so we abort the program.

4.1. On-line handling of violations

If we view the end-user as a process that invokes transactions, then violations raised by transactions and not handled by code written ahead of time are propagated up to the user. The traditional view of such a situation is that the transaction has failed and must be aborted. In light of our claim that it is impossible to anticipate all special cases in the program ahead of time, and the need to allow certain violations to persist, we allow the user to act as a violation-handler. This includes the ability to invoke existing transactions and to write code which is executed interpretively.

Note that this alone does not provide the user the same power for handling exceptions as that available to programmers. In particular, in the case of nested transaction invocations, programmers may attach handlers to any of the nested invocations, thereby allowing the higher transactions to continue without being interrupted. On the other hand, the user sees a violation only after the topmost transaction has raised it, and hence has only two choices: resume the entire computation, down to the initial signaller, or replace it all by some other action. If desired, one may provide more power to on-line handlers by displaying to the user the stack of activations that led to that violation, as in the INTERLISP system [28]; the user can then specify to which of these should his handler be attached.

5. RESUMING AFTER TYPE VIOLATIONS

Let us consider in detail the violation of a constraint that states that q-attribute values for instances of some entity or transaction class C must belong to some class R; such a constraint is marked by the presence of

$$q : R$$

on the definitions of class C. A violation of such a constraint is raised by the $MODIFY or $CREATE operation that attempts to assign a value to property q of some object b in C. To accommodate exceptions, we define $MODIFY and $CREATE in such a way that if they are **resume**d, the specified value will be stored as desired. In this case we say that <u>the value of b.q is exceptional</u>, or that "b.q is an exceptional fact", until some later update modifies this property value. Clearly, the use of such exceptional attributes can lead to problems. For example, if the value of q was supposed to be a number, but is a string now, then the statement

```
for z in C do
   z.q ← z.q +1;
   end
```

will cause a run-time error when z is b, even though it looked type-safe. These problems become even graver in the context of Information Systems, where many users *share* the database; unless we are careful, other users and programs will trip over such exceptional values since most likely they are not aware of their existence. We will therefore provide ways to mark exceptional attributes, and provide constructs that allow programmers to navigate around exceptional values.

5.1. Assigning exceptional property values

Whenever some property is to receive an exceptional value, we will require that before resuming, the user create an instance of the special class EXCEPTIONAL_PROPERTY (abbreviated EXNAL_PROP) described in Figure 8.

```
exception class EXNAL_PROP  with
  attributes
        obj: ANY_OBJECT
        pr: ANY_PROPERTY
  end;
```

Figure 8: The exception class EXNAL_PROP

Remembering that the range of *amountDue* for RESIDENTIAL_CUSTOMERs is 0.00 .. 200.00, we can then assign a higher value for customer *joe* as follows:

```
joe.amountDue ← 310.00 ;

        [ ANY_VIOLATION⟨cl=RESIDENTIAL_CUSTOMER,pr=amountDue⟩ =>

         begin
           create in EXNAL_PROP with  obj:joe, pr:amountDue;
           resume;
         end; ]
```

As with violations, we can gain additional power by creating subclasses of EXNAL_PROP, and then relying on users to categorize properly the exceptions that they permit to enter the database. For example, one can mark customers who have very high bills by creating the exception subclass VERY_HIGH_AMOUNT

```
exception class VERY_HIGH_AMOUNT  IsA EXNAL_PROP with
    attributes
        obj: CUSTOMER;
    end;
```

and then use VERY_HIGH_AMOUNT instead of EXNAL_PROP in the **create** statement of the handler above. In order to classify new kinds of unanticipated exceptions, users are allowed to define new subclasses of EXNAL_PROP at run-time. Assuming that the hierarchies of classes can be examined at run-time, such subclasses are one way in which users can communicate with others about shared exceptions.

5.2. Locating and accessing exceptional properties

The existence of objects in EXNAL_PROP is of use in several different ways: One can retrieve and examine its instances as with normal classes. For example, one can examine all objects with exceptional addresses by the following code fragment:

```
for x in EXNAL_PROP such that (x.pr = address) do
      (* examine x.obj *)
        . . .
```

One can also use this class to avoid or select objects depending on whether their attributes are exceptional:

```
for c in CUSTOMER such that   not EXNAL_PROP⟨obj=c, pr=amountDue⟩ do
        total ← total + c.amountDue;
        end;
```

Note that such defensive code can be written even before any exceptional facts have been uncovered, and hence can prevent some of the problems mentioned earlier.

If a program or user tries to examine the value of an exceptional attribute, a warning will be issued in order to signal unusual circumstances. In an anthropomorphic analogy, this warning is like a memory of some earlier painful occurrence -- the initial violation of the constraint. In our language, warnings are issued by raising exceptions. This has the advantage of allowing us to deal with exceptional facts through the exception handling mechanism that we have already defined. Rather than define a new category of violations, we use instances of EXNAL_PROP as exception objects. In other words, whenever an object’s property is evaluated by $EVALUATE, if it is exceptional then the instance of EXNAL_PROP representing this fact is raised as an exception. To actually obtain the exceptional attribute value, just resume this exception. For example, we can choose to ignore customers with exceptional names as follows:

```
for m in CUSTOMERS do
        write (m.name);    [EXNAL_PROP   =>    doNothing]
        end;
```

On the other hand, in calculating the total amounts due from customers, we can choose to also consider those exceeding the normal bounds as follows:

```
total ← 0;
for y in CUSTOMER do
   total ← total + y.amountDue;     [VERY_HIGH_AMOUNT   =>   resume]
   end;
```

When an exceptional property is updated, the object in EXNAL_PROP marking its exceptional nature is destroyed and the update is performed in the usual way. Of course, this may lead to the detection of new violations, which may be allowed to persist, etc.

In some cases, we may wish to use a different value of the exceptional attribute than the one stored. It is convenient to have for this purpose the command **resume with**. For example, if the handler specification

```
            [ VERY_HIGH_AMOUNT  =>  resume with 100000 ]
```

is used instead of the one in the previous example, the result will be to use 100000.00 for any exceptionally high *amountDue*. Therefore **resume with** replaces the the original retrieve command, and this is especially needed when exceptions are handled by users or default handlers. As another example, if we wanted to use 70% of such attribute values, we could use as exception handler the transaction in Figure 9.

```
transaction class PARTIAL with
    variables
        v : ANY
    actions
        v ← y.amountDue  [EXNAL_PROP  =>  resume];
        resume with (0.7 * v);
    end;
```

Figure 9: Resuming with a modified exceptional value

6. EXCEPTIONS TO INTEGRITY CONSTRAINTS - REFINEMENTS

In this section we reconsider the problems of resuming after integrity constraint violations in light of the need to control the occurrence of such exceptions and the new technique for marking exceptional facts that we now have at our disposal.

6.1. Administrative control of exceptions

Semantic constraints are usually used to enforce the policies of an enterprise and therefore their violation is often a matter of *contravening policy*. Such exceptions to policies should of course be controlled through protection mechanisms, including the standard techniques of database security. We should also maintain appropriate audit trails of such occurrences for later accountability. We introduce for this purpose *excuses*, which are objects in the special class

```
entity class ANY_EXCUSE with
   attributes
        viol: ANY_VIOLATION
        who: USER
        when: DATE
        expiry: DATE
        why: TEXT
   end;
```

We then restrict **resume** so that before we can resume after an integrity violation, an instance of

ANY_EXCUSE must have been created with that violation object as the value of its *viol* attribute.[8] The *who* attribute will be automatically set to be the user creating the excuse instance, and the *when* attribute will record the time when the excuse was created. The *why* property gives an opportunity for the excuser to provide an explanation of the situations surrounding the exception.

This allows us, for example, to find out the circumstances under which customers were permitted to exceed their credit limits -- this was checked by property *underLimit!* of RESIDENTIAL_CUSTOMER -- by executing

```
for v in ANY_VIOLATION
                such that (v.cl=RESIDENTIAL_CUSTOMER) and (v.pr=underLimit!)
  do
     get e in ANY_EXCUSE with viol=v;
     write( e.who, e.when, e.why);
  end;
```

The expiry date on an excuse can be used to allow temporary suspensions of constraints. The semantics of this attribute is that when the current date is past the expiry date of an excuse, the database administrator is warned by being given a violation to handle, namely the violation attached to the expired excuse. This mechanism can be combined with the familiar technique of defining subclasses of ANY_EXCUSE, in order to achieve varying powers for dealing with exceptions to constraints. For example, one can define the class TEMPORARY_METER_READ_EXCUSE as follows:

```
    entity class TEMPORARY_METER_READ_EXCUSE IsA ANY_EXCUSE with
       attributes
            viol: METER_READ_VIOLATION
            who: SECRETARY
       initially
            onlyOneDay: (expiry - when) < "2 days"
       end;
```

and then allow secretaries to create only such excuses, with the expectation that someone higher in authority will check the situation before expiry, and either make the excuse permanent or make changes that remove the violation.

6.2. Exceptional facts and integrity constraints

In some cases, a violation of a semantic integrity constraint may bring to light the fact that some object has an exceptional property or is an exceptional instance of some class. For example, consider the case when the familiar constraint that "managers earn more than their subordinates", expressed in our language as the EMPLOYEE property

earnsLess!: (salary < mgr.salary)

evaluates to false for employee *amanda*, whose *mgr* is *monica*. In such a situation, it might be the case

[8]For this reason, such violation objects will have to survive the termination of the handler, and be accessible until the excuse is deleted.

that

- *amanda* earns more than usual because she works overtime, and hence her *salary* is exceptionally high;
- *monica* earns less than usual because she is a part-time employee; thus her *salary* is exceptionally low;
- *monica* is only temporarily the manager of *amanda*; hence *amanda*'s *mgr* property is exceptional.

One or more such special cases essentially "explain" the failure of the integrity constraint; in other words, the failure can be *blamed* in this way on one or more exceptional facts. Although we have introduced exceptional properties as a technique for dealing with property values that violate attribute type constraints, we can use them more generally to mark any information which has to be interpreted "cautiously" or in a special way. Therefore exception handlers, in particular users, are allowed to create instances of EXNAL_PROP marking "blamed facts" after the violation of any constraint. This act of blaming is important for two reasons: it lets others know of the exceptional nature of the information uncovered, and it will permit us to reconcile the contradiction between the constraint and the databases, thereby allowing the constraint to continue fulfilling its role of error detection (see Section 7). Note that unlike earlier proposals for handling violations of semantic constraints, we do not assume that the latest update is exceptional when a violation occurs -- an update may bring to light earlier problems.

7. THE LOGIC OF ASSERTIONS WITH EXCEPTIONS

If a logical assertion has a violation that persists because it has been excused, then after every update the constraint will continue to be violated because the situation that has raised the first violation continues to hold. This means that the constraint is always raising violations, whether or not the latest update is correct, and the constraint has therefore lost its ability to detect errors -- its primary function. Furthermore, it has become a nuisance because we must continually excuse its violation and resume.

One solution to the second problem is to mark the property corresponding to the constraint as exceptional. For example, if some customer c violates his credit limit constraint, expressed by the CUSTOMER assertion property

underLimit!: (amountDue ≤ creditLimit)

then we can allow the situation to continue by making *underLimit*! exceptional for c:

create in *EXNAL_PROP* **with** *obj:c, p: underLimit!*

The semantics of this is to "turn off" the checking of the constraint *underLimit!* for object c. This is consistent with our view that if a property of c, such as *amountDue*, is marked exceptional, then we do not check the attribute type constraint for that object.

Unfortunately, this does not resolve entirely the problem of continuing to check future updates for validity. Consider for example the portion of the class definition for PERSON below:

```
entity class PERSON with
    attributes
        father: PERSON
        mother: PERSON
        age: YEARS
    always
        parentsAgeOK! : (father.age > 14) and (mother.age > 13)
        childrenAgeOK!: for every y in PERSON (y.mother=this ⇒ y.age < age)[9]
    end;
```

Suppose that some person *charlie* has an under-age father. By marking *parentsAgeOK!* exceptional for *charlie*, we stop checking this constraint for *charlie*; as a result, if by mistake someone later assigns *charlie* a mother who is under 13 years of age, this error will no longer be detected. Similarly, if *charlie* is allowed to have one child who is older by excusing *childrenAgeOK!*, then the constraint will no longer be checked on the ages of any of *charlie*'s children. In general, any assertion containing conjunction or quantifiers will suffer from this problem.

For this reason, we seek a mechanism that allows us to

- automatically modify an integrity constraint after one of its violations has been excused so that it appears consistent with the current database;
- at the same time ensures that the modified constraint continues to be as useful in detecting errors as it was before.

Such a mechanism is faced with the problem that in general there are many ways of restoring consistency in an inconsistent theory. We will take advantage of the presence of the user to ask him to point to one or more specific facts as "causes" of the violation, and then use this information to modify the constraint.

It will be convenient to switch to a logic framework for our database and constraints. In particular, we adopt a proof-theoretic approach resembling that in [23], where: each class is represented by a unary predicate; each property is represented by a binary predicate; all quantifiers are over a restricted range provided by some unary predicate; and a database state is a collection of ground atomic formulae and their negations. Using this syntax we can restate the assertion *parentsAgeOK!* into the form

$$\begin{array}{l}(\forall\ e,m,f/PERSON)\ (\forall\ a,b/YEARS)\\ \quad (FATHER(f,e) \wedge AGE(f,a) \Rightarrow (a>14))\ \wedge\\ \quad (MOTHER(m,e) \wedge AGE(m,b) \Rightarrow (b>13))\end{array}$$

A database Δ is defined to be a 3-tuple (V,D,C), where

- V is the set of unary and binary predicates allowed.
- D is a set of assertions specifying that properties are single-valued (functions), are applicable only to instances of classes for which they have been defined, and in each case have the desired range of values. For example, if class E has property definition *p:B*, then we need, among others, assertions $p(x,y) \wedge E(x) \Rightarrow B(y)$ and $p(x,y) \wedge p(x,z) \Rightarrow (y=z)$.
- C is a set of integrity constraints of the form described above.

[9]In English, this constraint states that all children are younger than their mothers. I am aware that there are easier ways to state this but this is one valid way of doing it.

A *database state* of Δ is a set F of "database facts": ground atomic formulae or their negations involving predicates in V and some finite set of constant symbols drawn from a fixed universe of symbols U. We make the usual "closed world assumption" about database states [23], so that for every predicate P and vector ω of constants, F contains either $P(\omega)$ or $\neg P(\omega)$. Queries to such a database state are then answered by finding proofs from the formulae in it.[10] A constraint in D or C is said to *hold* (or "be true") in a database state iff it is consistent with it.

In order to gain some insight into the problem of exceptions, consider for the moment a constraint Υ in conjunctive normal form. When Υ does not hold in some database state, then for at least one combination of values of the universally quantified variables, at least one conjunct is contradicted by the database. Since such a term is a disjunction of atomic formulae, possibly negated, this contradiction can be "*blamed*" on one or more of the predicates having the "wrong value" at their arguments. Thus if $\neg P(a) \vee Q(b)$ is contradicted by the current database then this can be blamed on either P(a) or $\neg Q(b)$ (or both) being currently true. (Note that in the previous section, we have asked the user to specify just such a set of exceptional facts whenever an integrity constraint was suspended.) If the blamed predicates would be changed to have the opposite truth-value at these arguments, then the constraint would again be consistent with the new database state, and hence could be used to detect errors. But the database state should not be modified - it accurately reflects reality; instead, we will modify the assertion to "simulate" this consistent state of affairs.

We formalize this line of reasoning as follows. Let Ψ be some constraint, and assume that it is violated in the current database state db. An integrity constraint Ψ for some database can be viewed as characterizing a set V_Ψ of database states: exactly those states consistent with it. Formally,

$$V_\Psi = \{ F \mid F \text{ is a database state of } \Delta \text{ consistent with } \Psi \}$$

Therefore, in the current situation db doesn't belong to V_Ψ. Suppose that the user blames this on the value of predicate P at argument β.[11] This means that Ψ is consistent with the database state db* that is identical to db except that $P(\beta)$ is "negated" in it. Formally, db* is defined as follows

Definition

If Φ is a ground atomic formula or its negation, then db* $\vdash \Phi$ iff db $\vdash \Phi$ unless $\Phi = P(\beta)$ or $\Phi = \neg P(\beta)$, in which case db* $\vdash \Phi$ iff db $\vdash \neg\Phi$. We use the notation **Switched**(db,P,β) to refer to the database state constructed from state db according to the above definition (i.e, db*=**Switched**(db,P,β)).

Our goal was to find a constraint Ψ^* that is consistent with the current database state (in which $P(\beta)$ is exceptional) but that detects errors in the same way as Ψ did. Given the above definitions, this is equivalent to finding a Ψ^* such that

[10]In proofs, we can make use of "unique name axioms" stating that all constant symbols are unequal -- see Reiter [23] for further details.

[11]We limit ourselves to blaming single facts, treating multiple blames as successive single blames. Also, we use lower case greek letters to refer to vectors of arguments for formulae.

$$V_{\Psi^*} = \{ \textbf{Switched}(D,P,\beta) \mid D \text{ in } V_\Psi\}$$

In other words, the database states consistent with Ψ^* are the same as those consistent with Ψ, except that the blamed predicate is "switched" at the argument where it is exceptional. Such a formula always exists and in fact can be obtained by syntactic manipulation, as indicated by the following theorem.

Theorem:

If db is a database state that is inconsistent with an integrity constraint Ψ, and this inconsistency can be blamed on the exceptional nature of some predicate *P* at argument β (i.e., **Switched**(db,*P*,β) is consistent with Ψ), then the formula Ψ^*, obtained from Ψ by replacing every occurrence of $P(\omega)$ for each argument ω by $(\omega\neg=\beta)\wedge P(\omega) \vee (\omega=\beta)\wedge\neg P(\omega)$, has the property that

1. Ψ^* is consistent with the current database state db;
2. $V_{\Psi^*} = \{ \textbf{Switched}(D,P,\beta) \mid D \text{ in } V_\Psi\}$

Proof[12] For any database state D, the information in **Switched**(D,*P*,β) can be represented by the introduction of a new predicate P^* that represents the "switching" of *P* at β. In particular, if we define the formula Π by

$$\Pi \equiv (\forall x)\ P^*(x) \Leftrightarrow (x\neg=\beta \wedge P(x)) \vee (x=\beta \wedge \neg P(x))$$

and then let $D^* = D \cup \{\Pi\}$, then we always derive the same information from D* as from **Switched**(D,*P*,β), as long as we consider P^* whenever *P* is mentioned in a query. But this means that $\Psi<P^*/P>$ will be consistent with D* iff Ψ was consistent with **Switched**(D,*P*,β). Furthermore, by our hypothesis, $\Psi<P^*/P>$ together with Π are consistent with our current database state db. Therefore we have achieved the reconciliation of the exceptional data with the integrity constraint at the cost of introducing a new predicate symbol and one new axiom for it. However, since the formula Π actually defines the predicate P^*, according to a theorem in logic on the elimination of definitions in proofs (e.g., [13]), we can uniformly substitute the right hand side of this definition throughout Ψ, to obtain Ψ^*, which is equivalent to $\Psi<P^*/P>$ in the sense that the conclusions derivable from $\Psi<P^*/P> \cup \Pi \cup D$ are identical to those derivable from $\Psi^* \cup D$. But this is exactly what was desired. *Q.E.D.*

Logical identities can be used to present the above transformation as two simpler transformations, applicable in complementary situations:

- if $P(\beta)$ is in the database state, then replace $P(\omega)$ by $(\omega\neg=\beta) \wedge P(\omega)$;
- if $\neg P(\beta)$ is in the database state, then replace $P(\omega)$ by $(\omega=\beta) \vee P(\omega)$;

Thus, for example, if the failure of the constraint *parentsAgeOK!* for PERSONs:

$(\forall$ e,m,f/PERSON$)$ $(\forall$ a,b/YEARS$)$
  $(\text{FATHER}(f,e) \wedge \text{AGE}(f,a) \Rightarrow (a>14)) \wedge$
  $(\text{MOTHER}(m,e) \wedge \text{AGE}(m,b) \Rightarrow (b>13))$

is blamed on the fact the *charlieSr*, the father of *charlie*, is too young (i.e., AGE(*charlieSr*,12) is exceptional), then the new constraint will be

$(\forall$ e,m,f/PERSON$)$ $(\forall$ a,b/YEARS$)$

[12]This simplified proof was suggested by R.Reiter.

$(\text{FATHER}(f,e) \wedge (\sim(f=\textit{charlieSr} \wedge a=12) \wedge \text{AGE}(f,a)) \Rightarrow (a>14))$
$\wedge\ (\text{MOTHER}(m,e) \wedge \text{AGE}(m,b) \Rightarrow (b>13))$

Contrast this with the result of excusing the *parentsAgeOK!* property of *charlie*, which would correspond to the formula

$(\forall\ e,m,f/\text{PERSON})\ (\forall\ a,b/\text{YEARS})$
$(e=\textit{charlie}) \vee$
$((\text{FATHER}(f,e) \wedge \text{AGE}(f,a) \Rightarrow (a>14)) \wedge (\text{MOTHER}(m,e) \wedge \text{AGE}(m,b) \Rightarrow (b>13))$

Note that whenever an exceptional fact is later updated, the constraint has to be modified back so that it considers the actual facts again. The transformation in our theorem is *commutative* -- i.e., the order in which things are blamed is not important -- and *idempotent* -- i.e., two successive blames of the *same* fact cancel each other out. (Both of these facts can be easily verified by considering the definition of **Switched**.) We can therefore easily cancel the exceptionality of a fact after an update by formally applying the same transformation which blamed it originally.

To summarize, we have provided a logical basis for reconciling an assertion with a database state that is inconsistent with it. This approach relies on having a subset of facts singled out as the cause of the contradiction. The presence of users in an Interactive System to provide such information is therefore important to the technique, although one can use the heuristic that blames the latest update for the contradiction.

8. SOME IMPLEMENTATION CONSIDERATIONS

There appears to be general agreement on the need for exception handling in modern programming languages, and in particular Taxis already has such facilities. We therefore do not consider here any problems connected with this issue alone (see [19] or [20] for implementation discussions).[13] Instead, we offer several techniques for dealing with the problems raised by allowing exceptional facts to enter the database through resumption -- the major novelty of the present proposal. In particular, we consider the costs of permitting violations of type constraints.

The implementation costs of our technique for allowing exceptional property values in an Information System can be categorized as follows:

1. Marking exceptional attributes and storing their actual values.
2. Determining when some particular attribute being accessed is exceptional, so that the exception can be raised by $EVALUATE.
3. Run-time type checking required to maintain type correctness. Type information about parameters and arguments is used by compilers to optimize procedure invocations by eliminating run-time checks of type compatibility. This applies to both user-defined transactions and primitive

[13]We have however tested the internal consistency of the exception handling mechanism proposed in Sections 3 and 4 by implementing a simulation of it in INTERLISP on a DEC-20. I am indebted to John Bresina for help with this implementation.

operators such as assignment and addition. Once we admit the possibility of resuming $EVALUATE for an exceptional property, such type inferences cannot be relied on any longer.

4. The apparent need to carry around type information together with every value. This is required in order to support the above run-time checks, since a value is usually just a bit-pattern that can represent an integer, characters or an object identifier.

In suggesting mechanisms for making the cost of exceptional values acceptable, we make use of the assumption that by definition exceptions are relatively rare occurrences, and that the cost of dealing with exceptional information can be considerably higher than the cost of dealing with normal information. We do not however wish to pay a major penalty for allowing the possibility of exceptions: i.e., normal information should be handled almost as efficiently (both in terms of time and space) as when there were no exceptions. In the remainder of this section, we offer several techniques that can be used to achieve the above goals. We assume that information about entities is stored in records (one or more records per entity) that have index structures resembling those in relational systems (see [7], for example).

To resolve the first two problems, exceptional property values can be stored in a separate file from those of normal values. The field of a record that normally stores the value of a property for an entity is set to **nil** when that property value is exceptional[14]. The actual value of the exceptional attribute is stored in a storage structure whose contents are described schematically by the relation (with variable length fields) EXCEPTIONAL_VALUE below.

EXCEPTIONAL_VALUE

Object	propertyName	valueType	value	exception

<u>key</u>: (Object,PropertyName)

Whenever the property of some object is retrieved and is found to be **nil**, the table EXCEPTIONAL_VALUE is consulted both in order to raise the appropriate exception, and in order to retrieve the actual value if the $EVALUATE operation is resumed (as it is likely to be). If no entry is found in the table, the attribute is assumed to be uninitialized. Ideally, the signalling of a **nil** value would be supported by hardware traps, as is division by 0, so that testing for **nil** does not require an extra CPU cycle. Such a scheme would resemble, for example, the technique used in some FORTRAN compilers on the IBM 7040 to detect access to uninitialized variables by using the the parity bit of memory words to mark uninitialized locations. If such hardware support does not exist, we can use horizontal splitting to reduce the need for run-time testing, as described below.

The above scheme allows us to store, retrieve and signal exceptional values and at the same time

[14]Nil is an uninterpreted special value.

preserve the advantages of strong typing for storage economy, especially known bounds on the space needed to hold an attribute, and the ability to dispense with type information for most values. We must still address the cost of run-time type checking, which appears to be needed because any attribute value could turn out to be exceptional, and hence evaluating any assignment, procedure call or expression could be problematic. For example, an exceptional *amountDue* could be a string for some reason, or just an integer greater than 200.

First, note that even currently compiled languages need some run-time checks. Such checks are needed during assignment in languages that, like Pascal, allow subranges of integers to be types. Also, in languages that allow general type hierarchies, such as Taxis, type checking needs to be done when a value known to be in some class C is assigned to a variable or attribute of type D, where D is a subclass of C. Finally, compiled languages (should) also perform run-time checks to detect un-initialized variables, or null values retrieved from the database. Therefore the issue is to what *degree* run-time type-checking can be eliminated.

The general approach to this problem which we have pursued is to provide both optimized and unoptimized versions of program fragments, and then set up situations where use of the optimized code is guaranteed not to allow type errors to slip by. Two examples of this technique follow.

First, when a transaction is invoked, we can start by being optimistic and executing a version of the transaction that has been optimized in standard ways. If a violation raised in that transaction, especially one signalled by $EVALUATE, is resumed, we can become "cautious" by undoing the current effects of the transaction, and then restarting an unoptimized version of the transaction, one which makes no assumptions and performs *all* run-time checks. The major cost of this mechanism is in the extra space required for keeping the unoptimized version of the transaction, since we need the interpreter to execute run-time handlers in any case. The cost of abandoning a partial computation and starting it over is incurred only when an exception is encountered, and is therefore acceptable if exceptions are indeed relatively rare occurrences. Note that this technique relies on $EVALUATE signaling an exception when an exceptional fact is accessed, and therefore cuing us when to switch to an unoptimized version of the code.

As a second example, consider loops. In a loop, such as

```
for y in CUSTOMER do
     total ← total + y.amountDue;
     end;
```

the likelihood of encountering an exceptional attribute is increased. And once we check that each *amountDue* retrieved is an integer, we incur a cost which is not bounded by a fixed values since it depends on the number of customers. To avoid this, we segregate instances of a class which do not have exceptional attributes from those that do. For example, we can partition the instances of every class C into two lists, U_C and E_C, such that all and only those objects in C having some exceptional attribute are held in E_C. (Note that these lists can be maintained as indices into some single file, or the exceptional tuples can in fact be stored in a distinct relation, which is called "horizontal splitting".) Using the fact that the language semantics specify that a **for**-loop traverses the instances of its class in non-deterministic order, we rewrite any **for**-loop over some class C as two successive iterations: one over U_C, the other over E_C. For the above example, these would be

```
for y in U_C do
        total ← total + y.amountDue;
    end;

for y in E_C do
        total ← total + y.amountDue;
    end;
```

The important point here is that in the loop over unexceptional instances, the code is optimized by assuming that all attributes of the loop variable meet their type specification; dealing with attributes of the loop variable in the second loop requires code for run-time checks. The improvement due to the above scheme should be significant since by our assumption, there should be many more objects in U_C than in E_C. This scheme has an added advantage in the case when there is no hardware support for detecting nil values: when accessing objects in U_C, we can use an optimized version of $EVALUATE which does not check if the value is nil.

If some class C has very many attributes, then even though there may be relatively few exceptional values for each attribute, the number of elements in E_C may become significant. This can be dealt with by a technique similar to vertical splitting of records: using external information (e.g., code analysis, IsA hierarchies, access statistics), attributes of a class C can be grouped into smaller sets A1, A2, ... so that the attributes of an object in Ai are often accessed together; such attributes are often stored together on the disk in order to reduce seek time ([7]). We then provide several pairs of indices for traversing C: U_A1_C and E_A1_C, U_A2_C and E_A2_C, etc. and the choice of which index to use for decomposing a particular loop would be made on the basis of which set of attributes occurs most frequently in the body of the loop.

We have therefore presented a simple technique for storing exceptional values in a database, which allows normal data to be stored and accessed as efficiently as usual. Note that the technique of segregating exceptional and unexceptional property values is quite useful for query processing as well as transaction optimization. We have also discussed several ways of avoiding the cost of additional run-time type checking in the case when no exceptional properties are evaluated. In general, the cost of the above solutions is paid in increased compile time (to be expected) and increased space for code.

9. FURTHER USES OF EXCEPTIONAL PROPERTIES

By marking information in the database as exceptional, we can use the exception handling mechanism to react to such information in a non-standard or context-dependent manner, including run-time decisions by users. The following are several examples of situations where this capability is useful. We give them as evidence that the concepts of exception handling and exceptional property value are independently motivated by other problems.

9.1. Null values and estimates

The value of an attribute in a database usually represents the up-to-date knowledge of its users. In cases when this knowledge is incomplete or inexistent, the attribute is said to have *"null value"*. Considerable research has been done on null values in databases (see [31] for some recent results), and most practical proposals share two related assumptions: i) null values are processed "automatically" in the DBMS by extending the semantics of primitive operations such as equality; ii) the different ways in which knowledge is allowed to be incomplete is usually very limited: "value exists but is unknown", "no value possible", or "total lack of knowledge".

The first assumption is unsatisfactory because it does not allow the user to process null values in a context-dependent manner: If the *creditRating* of a customer c is null, then the identity *(c.creditRating = high)* will be decided in one and only one way in all programs and queries; this certainly seems inappropriate. If these languages provided a test for null value, then we could achieve the desired effect by explicitly checking for null in every computation and comparison, but the resulting programs would become extremely cluttered. The issues raised here are very similar to those involving, for example, arithmetic underflow ("should 0 always be used?"), and not surprisingly we propose the same solution: explicit handling of exceptions. Thus attributes with null value should be marked exceptional by using a new subclass NULL of EXNAL_PROP. By our earlier mechanisms, whenever such an attribute is encountered, a NULL exception will be raised, and an exception handler will provide the desired behavior on a per case basis. For example, if we decide that in this context people with null *creditRating*s have low ratings, then we use the handler expression

[NULL⟨pr=creditRating⟩ => **resume with** low]

Elsewhere, we can use a different handler. Note that handlers can be provided either as part of the program/query, or by the user on-line, or by default.

The second assumption of current approaches to null value -- that there are only a few kinds of nulls -- is problematic because it severely limits the kinds of incomplete knowledge that we can capture. For example, we may know the value of a property but may not be able to represent it in the computer (overflow, say). Or we may have an estimate for the value, without knowing it exactly. Note that such knowledge is not irrelevant: knowing that some x.p is larger than the largest integer representable allows us to deduce that x satisfies the query condition (x.p ¬= 0), but not that it is equal to other values which are similarly large. These issues can also resolved in the framework of the above proposal to deal with null values through exceptional properties: Programmers and users can construct as fine a distinction between null values as they wish by creating appropriate subclasses of NULL, such as UNKNOWN_VALUE, NO_INFO, OVERFLOW, etc. Therefore, to specify that *joe*'s *amountDue* is not currently known, we state

create in UNKNOWN_VALUE **with** obj:joe, pr:amountDue

and to deal differently with UNKNOWN_VALUE and OVERFLOW nulls, we can specify alternate handlers as in

```
    (x.amountDue > 40000)   [UNKNOWN_VALUE  =>  return(false)]
                            [OVERFLOW  =>  return(true)]
```

Of course, it will be up to the users of this system to categorize actual property values appropriately, although one can set up in advance integrity constraints that ensure that the same object's property is not marked both UNKNOWN_VALUE and OVERFLOW for example.

The same approach can be taken to deal with other kinds of "informative nulls", such as estimates. These values are known not to be accurate reflections of reality (they may be out of date, or just projections) but are certainly not equivalent to total lack of knowledge. If for example we have a projected *amountDue*, we can define a new subclass, say FORECAST, of EXNAL_PROP or NULL, and store the value as follows:

```
            joe.amountDue ← 100.00;
            create in FORECAST with obj:joe, pr:amountDue;
```

This results in an exception being raised whenever *joe*'s *amountDue* is evaluated, in which case each particular user can decide whether to trust the stored value or not.

9.2. Measurements

We can also use exceptions to deal with a thorny issue in the area of representing measurements. A *measurement* is a (magnitude, unit) pair. Usually, each attribute representing a measurement has associated with it some particular unit so that only the magnitude needs to be stored. If the unit of a particular measurement is not the usual one, then storing just the magnitude will result in nonsensical computations. For example, suppose the salary attribute is stored in Dollars, but for some special employee it is in Francs; then storing the magnitude alone is not sufficient (since others will have no way of knowing that it is not in Dollars). It may also not be possible to convert it immediately to Dollars, because of fluctuating exchange rates: the same salary is to be paid each month in Francs, but the company's american balance will be debited different dollar amounts each time. Once again, the programmer can simply create FRENCH_FRANCS as a subclass of EXNAL_PROP, indicate that mary's salary is 4000 Francs by

```
    mary.salary ← 4000;
    create in FRENCH_FRANCS with obj:mary, pr:salary;
```

and, when necessary, convert this value to dollars through a handler similar to that seen in Figure 9; this handler could again be predefined, a default, or supplied by the user.

We emphasize that the actions described in the preceding subsections can be taken even after the Information System has been put in place, thereby supporting flexible handling of *unanticipated* exceptions.

9.3. Unexpected properties

In some circumstances, a particular object may have additional attributes to the ones specified in its class description. The simplest example of this is some sort of text comment field, which allows footnotes to be attached to an object. More interestingly, an object (e.g., a residential customer) may have a property normally restricted to another class (e.g., industrial customer), of which it is not an instance. Even more extreme is a case such as that of a customer with a foreign address. According to the definition in Figure 1, an address does not have a field for *country*, since addresses are assumed to be used locally only. Suppose a user is performing an on-line update to the database, and tries to store a *country* field with some particular object. By providing a subclass of EXNAL_PROP called EXTRA_PROPERTY, this can be accomplished by the following sequence of commands: the user first stores the part of the address that is not exceptional

```
    deGaulle.address ← [apt#:3, ...]
```

and then attempts to update the *country* attribute of this address

```
    deGaulle.address.country ← 'FRANCE'
```

When $MODIFY raises the exception INAPPLICABLE_PROPERTY, the user handles it by resuming:

```
            [INAPPLICABLE_PROPERTY⟨cl=ADDRESS,pr=country⟩ =>

             begin
               create in EXTRA_PROPERTY with
                    obj:thisViolation.obj,  pr:country;
               resume;
             end;
```

In order to retrieve the value of the extra property, one simply **resume**s the $EVALUATE operation after it signals that the specified attribute is not applicable:

```
    write(deGaulle.address.country)
            [INAPPLICABLE_PROPERTY⟨cl=ADDRESS,pr=country⟩ => resume]
```

If we wanted to be sure that others realize that deGaulle's address has an extra property, we could of course mark *address* itself exceptional.

A similar technique can be used when, contrary to expectations, an object has more than one value for a property (e.g., a person with several addresses). To begin with, create a subclass of EXNAL_PROPERTY called MULTI_VALUED_PROPERTY. Now, if for example some customer *sundanceKid* has several addresses, then store one of these in the *address* property, mark it exceptional by creating *(sundanceKid,address)* an instance of MULTI_VALUED_PROPERTY, and store the other addresses in new attributes *address1*, *address2*, etc. in the manner described above. A much cleaner solution to this problem is possible in languages allowing set or sequence-valued properties, such as ADAPLEX or Galileo, since here multiple values for single-valued properties are just type violations.

9.4. Exceptional Instances of Classes.

The data model supported by our language stores information subject to change in only two ways: through property values of objects, and through the instance relation between entities and classes. It is sometimes useful to specify that an object is an exceptional instance of some class, and not just that it has exceptional property values. Such situations arise when an object of one kind (e.g., a person) is considered to be an object of a different kind (e.g., a company) for the purposes of certain operations (e.g., taxation, electricity usage), but not others (e.g., having an accounting department), or in dealing with "hybrid" objects, like mermaids.

To deal with this situation, provide a second class, EXCEPTIONAL_INSTANCE, whose definition and use parallels that of EXNAL_PROP:

```
exception class EXCEPTIONAL_INSTANCE with
    attributes
        obj: ANY
        cl: ANY_CLASS
    end;
```

If for example stevenJob is a residential customer, then he can be made an exceptional instance of the class INDUSTRIAL_CUSTOMER through the statement

```
create in EXCEPTIONAL_INSTANCE with obj:stevenJob, cl:INDUSTRIAL_CUSTOME
```

When y takes on the value stevenJob in executing the loop

```
for y in INDUSTRIAL_CUSTOMER do . . .
```

an exception is raised -- namely the object in EXCEPTIONAL_INSTANCE created above. A handler is now required to decide whether this customer is to be treated the same as the others in this loop (i.e., resume) or whether he requires some special specified handling.

10. DISCUSSION

In the following sections we consider how the violation handling mechanism we propose fulfills some of our original goals.

10.1. Supporting the deferment of special cases

Our notation for violation handling and its semantics support the abstraction of typical cases in several ways. First, the specification of handlers appears as an annotation to the code dealing with the normal situation, thereby allowing the reader to ignore these "footnotes" in a first reading. The result is more readable code. This supports the abstraction of normal cases during software maintenance, when programmers other than the ones who wrote the original code must read and understand it.

Second, assertions and exceptional properties provide a unique and recognizable source of exceptional occurrences. This limits the notion of "exceptional occurrence" familiar in programming languages so that exceptions cannot be used for other programming tasks such as message passing or signalling the normal termination of loops. As a result, considering all the assertions can lead the programmer to write code for those cases that represent foreseeable special cases. This feature supports a *systematic* refinement process, during which the details abstracted away in the initial pass are introduced. Note that the ability to recognize the *source* of all special cases -- failed assertions -- does not necessarily imply the ability to anticipate all possible exceptional situations: just knowing that addresses could be abnormal does not mean that we can know in advance all possible forms of addresses to be encountered. Such unanticipated situations form one important source of special cases that must be handled at run-time.

10.2. Dealing with unanticipated violations

We have suggested that users who invoke transactions can be viewed as "operators", and hence can provide exception handling code at run-time. This is a simple but effective solution to the problem of handling violations whose cause had not been anticipated. The cost of such a mechanism is the need for an interpreter -- which we have seen may be independently useful for implementing transactions that resume. The **resume** statement is particularly useful since it allows constraints to be suspended at the user’s discretion, by simply providing a handler that has one statement: **resume**. Among others, resume can then be used to allow exceptional facts to enter into the database. On-line handling of exceptions is also useful in specifying in a context-dependent manner how to react to such information when we encounter it later.

Readers familiar with [20] may argue that the **resume** statement is contrary to good software engineering practice because it requires one to break the *procedural abstraction* barrier: one needs to find out about the internals of the procedure in order to decide on future actions. We claim however that this is not the case in most circumstances. Consider resuming a transaction T after one of its preconditions or postconditions has failed. According to prevalent theories of program development, anyone invoking a procedure should be aware of its pre- and postconditions in order to be able to use it properly. Therefore the information about the constraints is already available to the immediate invoker of

T -- the user if T is a top level transaction. Consider next the case of violations raised by primitive operations based on the constraints stated on entity class definitions. According to our view of a database as a world model, these entities correspond to real-world objects about which we have knowledge, rather then being implementation-related data structures that need to be hidden. Furthermore, the constraints stated on them capture the semantics of the enterprise, which presumably is clear to the users of the IS. Therefore resuming after a violation does not violate procedural abstraction in these cases. This leaves those situations where a violation is propagated to the top from some operation that is internal to the original transaction. In such cases, we contend that the provision of a new violation handler is an act of programming.

10.3. Resuming transactions as programming

Resuming a transaction essentially extends the domain of values over which an operation is defined. We can distinguish two kinds of constraints: *policy* and *essential* constraints. Intuitively, policy constraints are those whose violation does not render programs invalid. Alternatively, essential constraints are those needed to establish the weakest precondition of a transaction with respect to some stated goal. Logical conditions associated with transactions and entity classes are largely policy constraints, while property type definitions are mostly essential constraints.[15] Resuming after a policy constraint violation is strictly a policy decision. Resuming after an essential constraint violation is an act of programming because the goals of the transaction cannot be met by the current actions: we must accept that either the goals are different, or we must provide modifications to the transaction, or both. Now, a transaction is presumably resumed because it is both possible and convenient to supply these modifications in the form of exception handlers. Otherwise, the user should have aborted the transaction and invoked another one (either predefined or written at that moment). For example, suppose there is a transaction ENROLL(s,c) for enrolling a student in courses, which checks that the course is not over-enrolled, that the student has taken the prerequisites, etc., and then adds the student to the class roster, increments the class size by one, and adds the course to the student's record. Suppose that at some point a non-student (e.g., a secretary) wishes to enroll in a course. The person in charge must decide whether this is allowed, and if so, what variant of the procedure for enrolling students should be followed. Note that both these decisions are made on entirely external (i.e., not computer-related) grounds. Invoking the ENROLL transaction in this case will raise a number of violations along the way: the type of the s parameter is not STUDENT, and after we resume this, in several places s will be missing attributes associated with students (s does not have prerequisite courses, s does not have a student record to which the courses can be added, etc.). On the other hand, all manipulations of the course c by ENROLL are acceptable. If by handling each violation appropriately it is possible to arrive at the desired result, then this is a

[15]Integer subranges, for example, combine both aspects and thus blur the distinction.

reasonable way of defining a variant of enrolling into courses. If there are too many such violations (a subjective decision), then it would be easier to first write explicitly a new variant of ENROLL and then invoke it. In summary, **resume** provides an easier and more convenient way of defining *simple* variants of a transaction than the normal process of application program writing.

Note that resuming a transaction after encountering an exceptional property is an act of programming similar to the ones described above. Observe also that programmers can write code that anticipates and pre-empts such interruptions by filtering out exceptional facts (see Section 5). For example, a programmer may specify that certain operations on the attribute *p* should not be performed if this attribute is marked by the subclass SPECIAL_p of EXNAL_PROP; users would then be expected to classify each exceptional *p* as it enters the database. The result is software which deals with unanticipated cases gracefully -- our stated goal.

11. CONCLUSIONS

We have presented a coherent mechanism for exception handling in object-oriented languages for Information System design. We paid particular attention to the problem of accommodating violations of constraints such as semantic integrity constraints and type constraints associated with the database schema. The following are some of the significant contributions and key ideas of this work.

We have adopted the view that the various constraints imposed in an IS are normalcy conditions, and then provided the **resume** command to allow violations to such constraints to persist. Information considered to be the cause of a violation, whether of a type or integrity constraint, was marked in order to allow other users of the database to "navigate" around it. We have shown that the exception handling mechanism can in fact be used for two purposes:

- to detect and react when constraints are violated;
- to detect and react when exceptional facts are accessed.

To achieve this, the exceptional nature of certain facts is marked by the presence of objects in the class EXNAL_PROP, and these same objects are raised as exceptions when the exceptional facts are retrieved. Furthermore, resuming the interruption results in the actual exceptional value being returned.

We have also demonstrated the wider utility of the concept of an exceptional fact -- one which raises an exception when accessed -- by showing how it can be used to deal with a variety of problems in databases such as various null values, estimates, measurements, multi-valued or new attribute, and exceptional instances of classes. Two features of our language facility were crucial in resolving these problems, and have practical application elsewhere:

- the ability to define new exception types, and especially to organize them in a taxonomy;
- the ability to provide alternate actions depending on context.

The latter feature of exception handling mechanisms makes them considerably more powerful than previous techniques for reacting to special cases, such as triggers [9].

The mechanism we presented allows constraints to be suspended only in a controlled manner. This control is provided by

- the concept of excuses, which provides accountability;
- the restriction that an exception handler may not modify the values visible in the signalling transaction if it is resumed, which ensures that preconditions of transactions are not circumvented;
- the convention that if a transaction signals an exception and is not resumed, then its global effects are undone, so that the database does not remain in an inconsistent state.

The above techniques resolve the problems concerning the control, computation and sharing of exceptions to constraints in databases. We have also considered the implementation problems raised by persistent exceptions, and showed that by storing and indexing exceptional data separately, we can still use the standard techniques of data storage. We also mentioned ways of maintaining some of the advantages of compile-time optimization, by alternating the use of optimized and unoptimized versions of the code. Assuming that exceptions are relatively infrequent, they need not extract an unacceptably large computational toll.

Finally, we have given a logical foundation to the notion of "accommodating exceptions" by providing a technique for "minimally" modifying logical assertions so that they are again consistent with the database. This effectively maintains the advantages of integrity checking even in the presence of facts contradicting the constraint.

Taking the more general view of software development, we have identified in this work *special case deferment* as an important abstraction principle for developing application software, and *exception specification* as the refinement process supporting this abstraction. The ways in which our particular mechanism supports this abstraction were described in Section 10.1. Also, we resolved the problem of being unable to anticipate all exceptional situations at the time programs are written through the simple technique of allowing "on-line" handling of exceptions. This permits users to provide minor variations of existing transactions as the need arises, and is particularly useful for admitting and reacting to "exceptional facts" in the database, as well as suspending integrity constraints.

In summary, we have shown that more flexible information systems can be developed by providing an exception handling mechanism that, among others, allows users to provide handlers at run-time, permits exceptions to be resumed, and copes with aberrant data by raising exceptions. Our current research considers a number of additional problems relating to exceptions: contradictions arising during the design phase itself, as in the case of a subclass definition contradicting the specification of one of its superclasses; the logic of general exceptions in databases of logical formulae; exceptions to constraints on views; and the possibility of using exceptions to suggest improvements to the Information System schema. We are also considering the protection issues raised by run-time specification of exception handling code, by studying the *in-vivo* evolution of programs allowed by the Darwin system [21].

Acknowledgments I have greatly benefitted from the comments and encouragment of numerous colleagues, and the timely and detailed suggestions of the referees

References

[1] Albano, A., L.Cardelli and R.Orsini.
Galileo: a strongly typed, interactive conceptual language.
Technical Report TR 83-11271-2 (Murray Hill, NJ), Bell Laboratories, July, 1983.
(To appear in ACM TODS).

[2] Bernstein, P.A., Blaustein, B.T. and Clarke, E.M.
Fast maintenance of semantic integrity assertions using redundant aggregate data.
In *Proc. of 6th Int. Conf. On VLDB*, pages 126-136. Montreal, P.Q., October, 1980.

[3] Bobrow,D.G. and B. Wegbreit.
A model and stack implementation of multiple environments.
Communications of the ACM 16(10):591-603, October, 1973.

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

[5] Brodie, M.L.
On modelling behavioural semantics of databases.
In *Proc. of 7 Int. Conf. On VLDB*, pages 32-43. Cannes, France, September, 1981.

[6] Buneman, O.P. and Clemons, E.K.
Efficiently monitoring relational databases.
ACM TODS 4(3):368-382, September, 1979.

[7] 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. of 8th Int. Conf. On VLDB*. Mexico, September, 1982.

[8] Cristian, F.
Exception handling and software fault tolerance.
IEEE Transactions on Computers C-31(6):531-540, June, 1982.

[9] Eswaran, K.
Specifications, implementations and interactions of a trigger subsystem in an integrated data base system.
Technical Report RJ 1820, IBM Research, San Jose, August, 1976.

[10] Eswaran, K.P. and Chamberlin, D.D.
Functional specifications of a subsystem for data base integrity.
In *Proc. of Int. Conf. On VLDB*, pages 48:67. Framingham, Mass., September, 1975.

[11] Goodenough, J.B.
Exception handling: Issues and a proposed notation.
CACM 18:683-696, December, 1975.

[12] Gray, J. et al.
The recovery manager of the System R Database Manager.
ACM Computing Surveys 13(2):223-242, June, 1981.

[13] Grzegorczyk, A.
An outline of mathematical logic.
Reidel Publishing Co., 1974.

[14] Hammer, M. and McLeod, D.
Semantic integrity in a relational database system.
In *Proc. of Int. Conf. On VLDB*, pages 25-47. Framingham, Mass., September, 1975.

[15] Ichbiah, J.D. et al.
Rationale for the Design of the ADA Programming Language.
ACM SIGPLAN Notices 14(6), June, 1979.

[16] Kent, W.
Limitations of record based information models.
ACM TODS 4(1):107-131, March, 1979.

[17] Koenig, S. and Paige, R.
A transformational framework for the automatic control of virtual data.
In *Proc. of 7th Int. Conf. On VLDB*. Cannes, France, September, 1981.

[18] Kunin, J.S.
Analysis and specification of office procedures.
PhD thesis, MIT, February, 1982.

[19] Levin, R.
Program sturctures for exceptional condition handling.
PhD thesis, Carnegie-Mellon University, June, 1977.

[20] Liskov, B.H. and Snyder, A.
Exception handling in CLU.
IEEE Trans. Software Eng. SE-5(6):546-558, November, 1979.

[21] Minsky, N. and A. Borgida.
The Darwin Software Development Environment for Evolving Systems.
In P. Henderson (editor), *Proceedings of ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Pratcical Software Development Environments*, pages 89:95. ACM SIGPLAN Notices Vol.19, No.5, April, 1984.

[22] Mylopoulos, J., Bernstein, P.A. and Wong, H.K.T.
A language facility for designing interactive database-intensive systems.
ACM TODS 5(2):185-207, June, 1980.

[23] Reiter, R.
Towards a logical reconstruction of relational database theory.
In M.Brodie, J.Mylopoulos and J.Schmidt (editors), *On Conceptual Modelling*, pages 191-233. Springer Verlag, 1984.

[24] Schwartz, S.P. (editor).
Naming, Necessity and Natural Kinds.
Cornell University Press, 1977.

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

[26] Smith, J.M., Fox, S. and Landers, T.
Reference Manual for ADAPLEX.
Technical Report CCA-81-02, Computer Corp. of America, January, 1981.

[27] Stonebraker, M.
Implementation of integrity constraints and views by query modification.
In *Proc. ACM SIGMOD Conf. Management of Data*. May, 1975.

[28] Teitelman, W.
INTERLISP Reference Manual
4th edition, XEROX PARC, 1974.

[29] Wasserman, A.I.
Procedure-oriented exception-handling.
Technical Report 27, Medical Information Science, University of California, San Francisco, February, 1977.

[30] Yemini, S. and D.M.Berry.
A modular verifiable exception-handling mechanism.
ACM Trans. Prog. Languages and Systems 7(2):214-243, April, 1985.

[31] Zaniolo, Carlo.
Database relations with null values.
Journal of Computer and Systems Science 28(1):142-166, February, 1984.

Table of Contents