A Language Facility for Designing Database-Intensive Applications

JOHN MYLOPOULOS
University of Toronto
PHILIP A. BERNSTEIN
Harvard University
and
HARRY K. T. WONG
IBM Research Laboratory

TAXIS, a language for the design of interactive information systems (e.g., credit card verification, student-course registration, and airline reservations) is described. TAXIS offers (relational) database management facilities, a means of specifying semantic integrity constraints, and an exception-handling mechanism, integrated into a single language through the concepts of *class*, *property*, and the *IS-A* (generalization) *relationship*. A description of the main constructs of TAXIS is included and their usefulness illustrated with examples.

Key Words and Phrases: applications programming, information system, relational data model, abstract data type, semantic network, exception handling
CR Categories: 3.70, 3.73, 4.22, 4.29, 4.33, 4.34, 4.39

1. INTRODUCTION

1.1 Motivation

A primary goal of database management is the reduction of software costs by promoting data independence. In the database literature, practical aspects of the development of applications software that use a database system are often treated as peripheral to the main thrust of database research. Until recently, applications programming has usually been considered in the context of a data sublanguage embedded in a conventional applications programming language. Some of the better examples of this approach include papers by Date [5] and Schmidt [17].

This work was supported in part by the National Science Foundation under Grant ENG77-05720, in part by the National Research Council of Canada, and in part by the Division of Applied Sciences at Harvard University.
Authors' addresses: J. Mylopoulos, Department of Computer Science, University of Toronto, Toronto, Ont., M5S 1A7, Canada; P.A. Bernstein, Aiken Computation Laboratory, Harvard University, Cambridge, MA 02138; H.K.T. Wong, IBM Research Laboratory, San Jose, CA.

A more recent trend is to design the programming language with database facilities as a single unit [16, 22]. This paper takes a step along this path by presenting an applications programming language that tightly integrates data with the procedures that use it (in the style, say, of SIMULA [4]).

Our language, called TAXIS,[1] is designed primarily for applications systems that are highly interactive and make substantial use of a database. These applications, which we call interactive information systems (IIS), are characterized by their handling of large volumes of transactions that are short, of predictable structure, and update intensive. Examples include credit card verification, student-course registration, and airline reservations. By applying our tools to a more limited domain, we can customize them to the domain. Also, by defining our problem more narrowly than that of "applications systems," it will be easier to evaluate the efficacy of our approach.

In the future, we see TAXIS at the center of a programming system that would permit a designer to interactively build an IIS with the help of specialized text-editing and graphics facilities. The system would include a relational database management system (DBMS). The DBMS provides an interface into which the database operations of the IIS can be compiled.

1.2 Design Principles

TAXIS is eclectic, combining concepts from three areas of computer science research: artificial intelligence (AI), programming languages, and database management. From AI we have used the concept of semantic network for data and procedure modeling [2, 11]. From programming languages we have borrowed the concept of abstract data type [12, 18] and exception handling [21]. Finally, from database management we have built on the concept of a relational database [8].

These ideas are married to form a concise language framework, yielding a novel and powerful collection of facilities. First, the semantic network modeling constructs represent a qualitative improvement in abstraction mechanisms over conventional programming languages. Database operations can work on hierarchies of objects, instead of independent tuples and relations (similar to [20]). Data can thereby be manipulated at varying levels of abstraction. We extend our semantic structures beyond relations and apply them equally to procedures, integrity constraints, and exception handling.

Second, by associating operations with the data they use, the semantics of the *database* can be represented in the applications program. This is in contrast to the sharp distinction between DDL and DML in most database languages. The semantic information can be used by the compiler to solve many integrity, security, and concurrency problems at compile time.

Finally, since the application is described in a formal semantic model, "meta-level" commands allow the application description itself to be manipulated by programming language commands. This permits database administrator functions to peruse the logical design on-line.

Four principles guided much of the TAXIS design:

(1) The language must offer relations and associated operations for database

[1] Taxis (τάξις): Greek noun meaning order as in "law and order" or class as in "social class," "university class," etc.

management, transactions for the specification of application programs, and exception-handling facilities to enhance the development of interactive systems.

(2) Each conceptual object represented in the language must have associated semantics that involve both a behavioral and a structural component. These semantics are expessed in terms of the notions of class, property, and the *IS-A* hierarchy (cf. "generalization" in [20]).

(3) As much of the language as possible should be placed into the framework of classes, properties, and the *IS-A* hierarchy.

(4) The schema (i.e., the collection of classes, along with their properties and the associated *IS-A* hierarchy) should be compilable into a language such as Pascal, enriched with a relation data type and associated operations (as proposed, for instance, in [17]).

The first principle is a consequence of the intended scope of the language. The second reflects our belief that much of the difficulty of designing and implementing IISs (usually translated into high costs of initial implementation and maintenance) is due to the lack of appropriate programming constructs in "conventional" languages (e.g., Cobol and PL/I) for handling the semantics of any one application. The third and fourth principles are the results of our concern for linguistic uniformity and efficiency. We consider both of them quite important given the multiplicity of sources of ideas and the complexity of the problem at hand.

Section 2 of the paper discusses the basic entities that constitute a TAXIS program. Section 3 describes the *IS-A* hierarchy as an organizational principle (abstraction mechanism) for the classes constituting a program. In Section 4 we present more details about the different categories of classes. Concluding remarks and directions for further research appear in Section 5.

The presentation of the language is rather informal and necessarily sketchy due to space limitations. The interested reader is referred to [14, 25] for more details.

2. OBJECTS AND PROPERTIES

There are three types of objects in TAXIS: *tokens*, which represent constants; *classes*, which describe collections of tokens; and *metaclasses*, which describe collections of classes.

2.1 Tokens and Classes

Tokens are the constants of a TAXIS program. For example, *john-smith* (representing the particular person called John Smith), '*SMITH,JOHN,B*' (representing the string SMITH,JOHN,B), and 7 (representing the number 7) are all tokens. Tokens are denoted throughout the paper by identifiers in lowercase letters and numerals; strings are delimited by single quotes.

A class is a collection of tokens sharing common properties. If a token t is an element of the collection associated with a class C we say that *t is an instance of* C. It may be helpful for the reader to compare TAXIS classes with SIMULA classes or programming language types as points of reference.

Some sample classes are *PERSON*, whose instances are tokens such as *john-smith*, representing particular persons, *PERSON-NAME*, whose instances are (string) tokens, such as '*SMITH,JOHN,B*' that can serve as proper names, and *INTEGER*, whose instances are integers such as 7. We use identifiers in uppercase letters to denote classes.

We call the collection of all tokens which are instances of a class *C the extension of C.*

2.2 Properties

Classes and tokens have properties through which they can be related to other classes and tokens. Some of the properties that may be associated with the class *PERSON* represent the following information:

> "each person has a name, an address, an age, and a phone number"
> "each person's name consists of a first and last name and possibly a middle initial"

For tokens, properties represent specific facts rather than abstract rules such as those presented above. Thus, *john-smith* will have properties expressing facts such as

> "*john-smith*'s name is '*SMITH,JOHN,B*', his address is 38 Boston Dr., Toronto, his age is 32, and his telephone number is 762-4377"

Properties are triples consisting of one or more *subjects*, an *attribute*, and a *property value* (or *p-value*). For example, *PERSON* may have the following properties:

⟨*PERSON*, *name*, *PERSON-NAME*⟩

⟨*PERSON*, *address*, *ADDRESS-VALUE*⟩

⟨*PERSON*, *age*, *AGE-VALUE*⟩

⟨*PERSON*, *phone#*, *PHONE-VALUE*⟩

The same applies for properties of tokens, i.e.,

(*john-smith*, *name*, '*SMITH,JOHN,B*')

(*john-smith*, *address*, *john-smith's-address*)

(*john-smith*, *age*, 32)

(*john-smith*, *phone#*, 7624377)

Note that the properties of *PERSON* provide information about the structure of instances of that class, while the properties of *john-smith* specify the structure of the token itself. This distinction was already made in the notation just introduced for properties, with the properties of a class delimited by angular brackets and those of a token by parentheses. We call the former type of property *definitional* and the latter *factual.*

Some properties may have more than one subject. For example,

⟨(*FLIGHT#*, *DATE*), *flt*, *FLIGHT*⟩

defines a (definitional) *complex property* with subjects the classes *FLIGHT#* and *DATE* and *p*-value the class *FLIGHT*. This property may represent the information:

> "each combination of a flight number and a date has an associated flight"

As the reader may have suspected, there is a strong relationship between the definitional properties of a class and the factual properties of its instances. The relationship may be expressed in terms of the following property induction principle.

Property Induction Principle. The definitional properties of a class induce factual properties for its instances.

If classes $C_1, \ldots, C_n$ are the subjects of a definitional complex property with attribute p, the TAXIS expression $(C_1, \ldots, C_n) .. p$ (or $C_1 .. p$ if $n = 1$) returns the p-value of that property. For example, *PERSON* .. *age* returns the class *AGE-VALUE*, while (*FLIGHT#*, *DATE*) .. *flt* returns *FLIGHT*. In other words, " .. " is a "schema selector" and allows the traversal of the schema defined with a TAXIS program by its classes and their definitional properties. For the " .. " operator to be unambiguous, no two definitional properties can have the same subject(s) and attribute.

Turning to factual properties, if $\langle (C_1, \ldots, C_n), p, C \rangle$ is a definitional property and t_1 is an instance of C_i, $1 \le i \le n$, then $(t_1, \ldots, t_n).p$ (or $t_1.p$ if $n = 1$) evaluates to an instance of C, say t, such that $((t_1, \ldots, t_n), p, t)$ is a factual property. Thus *john-smith.age* returns 32 while (802, *may*-1-1979).*flt* returns the particular flight associated with those two tokens through the *flt* property (i.e., the property with attribute "*flt*").

2.3 Metaclasses

If one wishes to represent the information

"the average age of (known) persons is 28"

or

"the number of (known) flights is 473"

he may be tempted to express these facts by

⟨*PERSON*, *average-age*, 28⟩

⟨*FLIGHT*, *cardinality*, 473⟩

However, this representation is incorrect since definitional properties represent information about the structure of instances of a class, not the class itself. Instead, factual properties must be used to represent these facts:

(*PERSON*, *average-age*, 28)

(*FLIGHT*, *cardinality*, 473)

But to be consistent with the property induction principle, these factual properties must be induced by definitional properties which have the classes *PERSON* and *FLIGHT* as instances. This observation leads to the introduction of a third type of TAXIS object called *metaclass*. A metaclass is similar to a class in every respect, except that its instances are classes rather than tokens. For instance, the metaclass *PERSON-CLASS* may be defined with instances of all classes whose instances denote persons (e.g., *PERSON*, *STUDENT*, *EMPLOYEE*, *MANAGER*). Then the definitional property

⟨*PERSON-CLASS*, *average-age*, *AGE-VALUE*⟩

allows the association of an *average-age* factual property with every instance of *PERSON-CLASS.*

(*PERSON, average-age*, 28)

(*STUDENT, average-age*, 19), etc.

We refer to the relationships between a token (class) and the class (metaclass) it is an instance of as the *INSTANCE-OF* relationship.

Generally, a TAXIS program includes tokens which can only have factual properties associated with them, classes which can have factual and definitional properties, and metaclasses which can only have definitional properties. For a more sophisticated treatment of the *INSTANCE-OF* relationship which allows an arbitrary number of levels of metaclasses, see [11] and [19]. We expect that the three levels allowed in TAXIS will suffice for most practical situations.

For metaclasses, we use identifiers in uppercase letters which end in *-CLASS.* As with classes, the collection of all instances of a metaclass is called its extension.

2.4 Examples

Classes and metaclasses are defined by specifying their name and their simple properties. For example, the metaclass *PERSON-CLASS* can be defined by

metaclass *PERSON-CLASS* **with**
  attribute-properties
    average-age: *AGE-VALUE*;
end

Here *PERSON-CLASS* is defined to have one simple (i.e., noncomplex) property

⟨*PERSON-CLASS, average-age, AGE-VALUE*⟩

The metaclass definition also specifies that the property defined is of the **attribute-property** category which means that the *average-age* factual property of an instance of *PERSON-CLASS* may change with time. Generally, every definitional property defined in a TAXIS program is classified into a unique *property category* at the time of its definition, which determines the functional and operational characteristics of the property.

Property categories allow the specification of information such as that the function defined by a property is time varying or 1–1 or should be used in a particular manner when instances of its subject(s) are created. The following examples illustrate the different uses of property categories.

The class *PERSON* can now be defined as an instance of the metaclass *PERSON-CLASS* by

PERSON-CLASS PERSON **with**
  keys
    person-id: (*name, address*);
  characteristics
    name: *PERSON-NAME*;
    address: *ADDRESS-VALUE*;
    phone#: *PHONE-VALUE*;
  attribute-properties
    age: *AGE-VALUE*;
    status: *STATUS-IN-CANADA*;
  end

According to this definition, *PERSON* has two attribute (i.e., time-varying) properties and three characteristic properties which are time invariant. The key property described in the definition of *PERSON* specifies the complex property

$$\langle (PERSON\text{-}NAME, ADDRESS\text{-}VALUE), person\text{-}id, PERSON \rangle$$

Thus ('*SMITH, JOHN, B*', *john-smith's-address*)*.person-id* returns the person with '*SMITH, JOHN, B*' as name and *john-smith's-address* as address, if any. If there is none, the expression returns the special TAXIS token **nothing**.

The class *FLIGHT* can be defined in a similar fashion:

VARIABLE-CLASS FLIGHT **with**
  keys
    flt: (*flight#*, *date*);
  characteristics
    flight#: {| 1 :: 999 |}
    departure: [| *city*: *CITY*, *country*: *COUNTRY* |];
    destination: [| *city*: *CITY*, *country*: *COUNTRY* |];
    aircraft: *AIRCRAFT-TYPE*;
    date: *DATE-VALUE*;
  attribute-properties
    seats-left: *NONNEGATIVE-INTEGER*;
end

Here *VARIABLE-CLASS* stands for a special metaclass whose instances can have their collections of tokens changed in terms of explicit insertions or removals. Thus, since *FLIGHT* is an instance of *VARIABLE-CLASS*, it can have tokens added to or removed from its collection of instances. Clearly, variable classes behave very much like relations [3]. *PERSON* can also be made an instance of the metaclass *VARIABLE-CLASS*, in addition to its being an instance of *PERSON-CLASS*, by relating the metaclasses *PERSON-* and *VARIABLE-CLASS* through the *IS-A* relationship. This is discussed in more detail in Section 3.

The class defined by {| 1 :: 999 |} is *finitely defined* in the sense that it has a finite, time-invariant collection of instances which includes all integers from 1 to 999. Since this class does not have an associated name, it can only be referenced through expressions such as *PERSON .. flight#*.

The class defined by [| *city*: *CITY*, *country*: *COUNTRY* |] has as instances all tuples with the first component an instance of *CITY* and the second an instance of *COUNTRY*. Classes such as this are instances of the special metaclass *AGGREGATE-CLASS*. Generally, an instance of *AGGREGATE-CLASS*, say *A*, has a collection of instances which is the cross product of the collections of instances of classes that serve as *p*-values of *A*'s characteristic properties. In this respect, aggregate classes are quite different from variable classes.

In other words, if aggregate class C has characteristic properties $p_1, \ldots, p_n$ with *p*-values $C_1, \ldots, C_n$, respectively, and if the extensions of these classes are $\text{ext}(C_1, \sigma), \ldots, \text{ext}(C_n, \sigma)$ in some database state σ, then

$$\text{ext}(C, \sigma) = \text{ext}(C_1, \sigma) \times \text{ext}(C_2, \sigma) \times \cdots \times \text{ext}(C_n, \sigma).$$

The class [| *city*: *CITY*, *country*: *COUNTRY* |] could have been defined separately.

```
AGGREGATE-CLASS LOCATION with
  characteristics
    city: CITY;
    country: COUNTRY;
end
```

with *LOCATION* replacing [| *city*: *CITY*, *country*: *COUNTRY* |]. If that second method were used,

$$FLIGHT..departure = FLIGHT..destination$$

With the original definition of *FLIGHT*, however, the above equality does not hold. In other words, each class definition that appears in a TAXIS program causes the introduction of yet another class in the schema described by the program.

Turning to some of the classes mentioned in the definitions presented so far, let us first define *PHONE-VALUE* as

```
FORMATTED-CLASS PHONE-VALUE with
  {| '(' |} @ REPEAT(DIGIT, 3) @ {| ')' |} @ REPEAT(DIGIT, 7)
end
```

Formatted classes (i.e., instances of *FORMATTED-CLASS*) have as instances all strings which are consistent with a given string pattern. In particular, *PHONE-VALUE* instances have the format '*(ddd)ddddddd*' where *d* is any digit. Here { | ')' | } defines a class with only instance the string ')', and *A* @ *B* defines a class with instances strings obtained by concatenating an instance of *B* to an instance of *A*. Moreover,

$$REPEAT(A, n) \equiv A \ @ \ A \ @ \cdots @ \ A \ (n \text{ times})$$

Finally, *DIGIT* is assumed to be the class { | '0', '1', . . . , '9' | }.

It was mentioned in the introduction that all TAXIS constructs are treated within the framework described so far. Thus transactions are classes too. For example, the transaction *RESERVE-SEAT* may be defined as follows:

```
TRANSACTION-CLASS RESERVE-SEAT with
  parameter-list
    reserve-seat: (p, f);
  locals
    p: PERSON;
    f: FLIGHT;
    x: INTEGER;
  prereqs
    seats-left? : f. seats-left > 0;
  actions
    make-reservation:
      insert-object in RESERVATION with
          person ← p, flight ← f;
    decrement-seats: f. seats-left ← f. seats-left − 1;
    assign-aux variable: x ← f. seats-left;
  returns
    rtrn: x;
end
```

The above definition specifies the parameter list of *RESERVE-SEAT* through the **parameter-list** property which defines a complex property

⟨(*PERSON*, *FLIGHT*), *reserve-seat*, *RESERVE-SEAT*⟩

Local properties (**locals**) define either parameters or local variables of the transaction. The body of the transaction is given in terms of zero or more prerequisite, action, and result properties (**prereqs**, **actions**, **result**, respectively) whose *p*-values are invariably expressions. Finally, the returns property (**returns**) associates with a transaction an expression to be evaluated when execution of the body of the transaction has been completed. The value of the expression is also the value returned by the transaction.

It is assumed in the definition of *RESERVE-SEAT* that *RESERVATION* has already been defined as a variable class and that it has two characteristics with attributes *person* and *flight*, respectively. Thus the **insert-object** expression inserts another instance into the extension of this class and sets its two characteristic properties to *p* and *f*, respectively. The other two action properties decrement the *seats-left* property of *flight f* by 1 and set the local variable *x* to the value to be returned by the transaction.

A transaction class is similar to a variable class in that it has a time-varying extension. When an expression involving a call to *RESERVE-SEAT* is evaluated, a new token is first created and added to the extension of *RESERVE-SEAT*. This token is essentially an execution instance of *RESERVE-SEAT*, and the factual properties associated with it indicate the values of local variables at any one time. In fact, for the expressions which appear inside the transaction, mention of a local variable or parameter, i.e., *p*, *f*, or *x* for *RESERVE-SEAT*, is interpreted as equivalent to *self.p*, *self.f*, *self.x*, where *self* denotes the execution instance with respect to which these expressions are evaluated. Something analogous applies to **prereqs**, **actions**, **result**, and **returns** properties which initially have *p*-value **unknown** (another special TAXIS token), until the corresponding expression has been evaluated. From that point on, the *p*-value of such a property is the value returned by the expression. Thus if the identifier *make-reservation* appears in an expression, before the *make-reservation* **action** property is evaluated its value is **unknown**, while after it is evaluated, it is the value returned by the **insert-object** expression.

As mentioned earlier, execution of a transaction begins by adding a token to the extension of the transaction (class). Execution then proceeds by evaluating each prerequisite *p*-value expression to make sure that it returns the value **true**. If any of the prerequisite expressions are found to have a value other than **true**, an *exception* is said to arise and execution is suspended. Otherwise, action expressions and then result expressions, which must also return **true** values, are evaluated. Thus prerequisite and result properties can be thought of as preconditions and postconditions which must be satisfied if execution of the transaction is to be meaningful. If they are not, an exception is raised and an exception-handling transaction is called to correct the situations. The exception-handling mechanism of TAXIS is discussed in Section 4.4.

When the *p*-value of a definitional property $\langle (C_1, \ldots, C_n), p, T\rangle$ is a transaction, the meaning of the property changes in that T specifies not the type of *p*-values of factual properties induced by $\langle (C_1, \ldots, C_n), p, T\rangle$, but rather an algorithm for getting them. For example, suppose the property

⟨*PERSON, birthdate, COMPUTER-BIRTHDATE*⟩

is added to the definition of *PERSON* where

TRANSACTION-CLASS COMPUTE-BIRTHDATE **with**
  parameter-list
    birthdate: (*p*);
  returns
    rt: *this-year* − *p.age*;
end

and *this-year* is an identifier that denotes the current year. Clearly, to every particular person this property associates not an instance of *COMPUTE-BIRTHDATE*, but rather a token returned by the *p*-value of the *rt* property.

This convention of treating transactions as a means for obtaining *p*-values rather than as types of *p*-values is consistent with the SIMULA class concept. Thus in TAXIS

p. birthdate ≡ *COMPUTE-BIRTHDATE*(*p*)

where *p* is an instance of *PERSON*. Similarly, for the parameter-list complex property associated with *RESERVE-SEAT*,

(*prsn, flt*).*reserve-seat* ≡ *RESERVE-SEAT*(*prsn, flt*)

3. THE *IS-A* HIERARCHY

We envision a TAXIS program as a large collection of tokens, classes, and metaclasses interconnected through their properties. Perhaps the most important feature of TAXIS is the facility it provides for organizing the collection of classes and metaclasses into a hierarchy (taxonomy).

3.1 Preliminaries

The *IS-A* (generalization) relationship is defined over classes and metaclasses. Informally, we say that (*A IS-A B*) where *A*, *B* are both classes (metaclasses) if every instance of *A* is an instance of *B*. For example, (*ADULT IS-A PERSON*) specifies that every adult is a person and (*CHILD IS-A PERSON*) that every child is a person.

If (*A IS-A B*) then every definitional property of *B* is also a definitional property of *A*. Moreover, *A* can have additional properties that *B* does not have at all, or it can redefine some of the properties of *B*. For example, the class *ADULT* inherits the *name*, *address*, and *phone#* properties of *PERSON* but must redefine the *age* property by restricting *age* *p*-values to instances of the class *OVER*-18. Similar remarks apply for *CHILD* which, in addition, has the *guardian* property that *PERSON* does not have at all. In defining the classes *ADULT* and *CHILD*, one need not mention the properties these classes share with *PERSON*:

VARIABLE-CLASS ADULT **is-a** *PERSON* **with**
  attribute-properties
    age: *OVER*-18;
end
VARIABLE-CLASS CHILD **is-a** *PERSON* **with**
  attribute-properties
    age: *UNDER*-18;
    guardian: *ADULT*;
end

Properties cannot be redefined arbitrarily. For example, redefinition of *age* only makes sense if (*UNDER*-18 *IS-A AGE-VALUE*). As the reader may have suspected, the *IS-A* relationship referred to above is the reflexive transitive closure of the relationship **is-a** used in class definitions.

3.2 *IS-A* Relationship Postulates

The formal properties of the *IS-A* relationship can be summarized in terms of the following postulates:

I. All classes (metaclasses) constituting a TAXIS program are organized into an *IS-A* hierarchy in terms of the binary relation *IS-A* which is a partial order.

II. There is a most general (maximum) and a most specialized (minimum) class with respect to *IS-A* called, respectively, *ANY* and *NONE*. Similarly, there is a most general and a most specialized metaclass called, respectively, *ANY-CLASS* and *NO-CLASS*.

III. (Extensional *IS-A* Constraint) If (C *IS-A* D) for classes (metaclasses) C and D, then every instance of C is also an instance of D.

IV. (Structural *IS-A* Constraint) If (A *IS-A* B) and B is the subject of a definitional property $\langle (C_1, \ldots, B, \ldots, C_n), p, D \rangle$, then A is also the subject of a definitional property $\langle (C_1, \ldots, A, \ldots, C_n), p, E \rangle$ and moreover (E *IS-A* D).

Note that these postulates define *necessary* not sufficient conditions for the *IS-A* relationship to hold.

It is assumed that there exist classes *ANY-FORMATTED*, *ANY-VARIABLE*, *ANY-TRANSACTION*, etc., which are specializations of *ANY* and below which one finds all formatted classes, variable classes, etc. For example, the definition given earlier

```
VARIABLE-CLASS FLIGHT with
    ...
end
```

places *FLIGHT* below *ANY-VARIABLE* and is therefore equivalent to

```
VARIABLE-CLASS FLIGHT is-a ANY-VARIABLE with
    ...
end
```

For metaclasses the *IS-A* hierarchy must be defined explicitly by the TAXIS user. For example, the metaclass *PERSON-CLASS* should be a specialization of *VARIABLE-CLASS*, as suggested in Section 2.4, and for this purpose its definition should be changed to

```
metaclass PERSON-CLASS is-a VARIABLE-CLASS with
    ... (as before)
end
```

After this change, all instances of *PERSON-CLASS* are also instances of *VARIABLE-CLASS* according to Postulate III, and therefore *PERSON* is a variable class.

The Hasse diagram of the *IS-A* relationship need not be a tree. For example, the definition

PERSON-CLASS MALE-STUDENT **is-a** *MALE, STUDENT* **with**
...
end

makes *MALE-STUDENT* a specialization of *MALE* and *STUDENT* which may not be *IS-A*-comparable.

The class *ANY* has as instances all tokens available to a TAXIS program, while *NONE* has no instances at all. Similarly, *ANY-CLASS* has all classes as instances, while *NO-CLASS* has no instances at all.

3.3 More on Seat Reservations

We return to the world of persons, flights, and seat reservations to illustrate the use of the *IS-A* hierarchy.

First, let us define a few specializations of previously defined classes.

INTERNATIONAL-FLIGHT# := { | 500 :: 999 | } **is-a** *FLIGHT* .. *flight#*
FLIGHT#-WITHIN-CANADA := { | 1 :: 499 | } **is-a** *FLIGHT* .. *flight#*

places the finitely defined classes with extensions the ranges 500 :: 999 and 1 :: 499, respectively, below *FLIGHT* .. *flight#* (= { | 1 :: 999 | }) on the *IS-A* hierarchy. Similarly,

CANADA := { | '*CANADA*' | } **is-a** *COUNTRY*

makes *CANADA* a class with a single instance. Presumably, *COUNTRY* has as instances many other strings such as '*USA*', '*CHINA*', and '*GREECE*', in addition to '*CANADA*'.

It is now possible to define two specializations of *FLIGHT*

VARIABLE-CLASS INTERNATIONAL-FLIGHT **is-a** *FLIGHT* **with**
  characteristics
    flight#: *INTERNATIONAL-FLIGHT#*;
end

VARIABLE-CLASS FLIGHT-WITHIN-CANADA **is-a** *FLIGHT* **with**
  characteristics
    flight# FLIGHT#-WITHIN-CANADA;
    departure: [| *country*: *CANADA* |] **is-a** *FLIGHT* .. *departure*;
    destination: [| *country*: *CANADA* |] **is-a** *FLIGHT* .. *destination*;
end

When a class is defined "on-line" in terms of the match-fix operators { | , | } or [| , |], one can place it at the same time on the *IS-A* hierarchy, as illustrated in the *departure* and *destination* properties of *FLIGHT-WITHIN-CANADA*. Of course, since the aggregate class defined by [| *country*: *CANADA* |] is a specialization of *FLIGHT* .. *departure* (= [| *city*: *CITY*, *country*: *COUNTRY* |]), it has two (not one) characteristic properties, as *city* is inherited.

According to the definition of *RESERVE-SEAT*, the definitional complex property

⟨(*PERSON*, *FLIGHT*), *reserve-seat*, *RESERVE-SEAT*⟩

is part of the TAXIS program being constructed. It follows then from Postulate IV (the structural *IS-A* constraint) that any combination of specializations of the classes *PERSON* and *FLIGHT* must have a *reserve-seat* complex property whose *p*-value, a transaction, is a specialization of the transaction *RESERVE-SEAT*. Intuitively, this means that the *reserve-seat* for, say, *CHILD*, and *INTERNATIONAL-FLIGHT* must have at least the prerequisites, actions, and results of *RESERVE-SEAT* and possibly more of each. For example, suppose that we wish to enforce a (rather conservative) constraint whereby each child must be accompanied by his/her guardian on an international flight. This is clearly a constraint concerning the transaction (*CHILD, INTERNATIONAL-FLIGHT*) . . *reserve-seat*. It can be added to that transaction as a prerequisite as follows:

prereq *accompanied-by-guardian?* **on**
  (*CHILD, INTERNATIONAL-FLIGHT*) . . *reserve-seat* **is**
  not ((*p.guardian, f*). *reservation* = **nothing**)

This definition adds *accompanied-by-guardian?* as a prerequisite property of the transaction (*CHILD, INTERNATIONAL-FLIGHT*) . . *reserve-seat*, which, of course, also inherits all properties of *RESERVE-SEAT*. The expression (*p.guardian, f*).*reservation* has value **nothing** when there is no instance identified by the key value (*p.guardian, f*) in the (variable) class *RESERVATION*; otherwise, it returns the instance of *RESERVATION* identified by that key value.

As another example, suppose that any person (adult or child) entering Canada must be a citizen, landed-immigrant, or visitor:

prereq *can-enter-canada?* **on**
  (*PERSON, INTERNATIONAL-FLIGHT*) . . *reserve-seat* **is**
  p.status **instance-of** {| '*CITIZEN*', '*LANDED-IMMIGRANT*', '*VISITOR*' |}
    or not *f.destination.country* = '*CANADA*'

As a final example of how specializations of *RESERVE-SEAT* might be modified to suit particular combinations of specializations of *PERSON* and *FLIGHT*, suppose that the income tax office must be notified for any citizens or landed immigrants leaving Canada:

action *notify-income-tax people* **on**
  (*ADULT, INTERNATIONAL-FLIGHT*) . . *reserve-seat* **is**
  if (*p.status* = '*CITIZEN*' **or** *p.status* = '*LANDED-IMMIGRANT*'
    and *f.departure.country* = '*CANADA*'
    and not (*f.destination.country* = '*CANADA*')
  then *NOTIFY-INCOME-TAX-PEOPLE*(*p, f*)

This action has no effects if its Boolean condition is not true.

Once these properties have been added to their corresponding transactions, the expression (*p, f*).*reserve-seat* has quite different meaning depending on whether *p* is an adult, a child, or just a person and *f* is an international or local flight. Generally,

$$(p, f).\textit{reserve-seat} \equiv (\textit{Type}(p), \textit{Type}\ (f))\ ..\ \textit{reserve-seat}(p, f)$$

where *Type*(*x*) returns (one of) the least general class that has *x* as an instance. If there is more than one such class, then it is assumed that choosing between them does not affect the value or the side effects caused by the call.

The examples presented illustrate the following points about the *IS-A* relationship.

(1) It is not only data objects that can be organized into an *IS-A* hierarchy but also semantic integrity constraints, expressed as prerequisites, results, and database actions.

(2) Parts of the *IS-A* hierarchy determine the structure of other parts through the definition of properties. For example, the part of the *IS-A* hierarchy which appears below the transaction *RESERVE-SEAT* is structurally homomorphic to the cross product of the *IS-A* hierarchies which appear below *PERSON* and *FLIGHT*. This is a direct consequence of Postulate IV (the structural *IS-A* constraint) and it can serve as a powerful guiding principle for the construction of a TAXIS program.

4. MORE ON CLASSES AND METACLASSES

We return to the topic of classes and metaclasses in order to provide additional details about them.

4.1 Variable Classes

The built-in metaclass *VARIABLE-CLASS* has the special feature that only its instances can have their extensions altered through the expressions **insert-object, remove-object**. For example,

VARIABLE-CLASS PASSENGERS **with**
  p: *PERSON*
end

defines an instance of *VARIABLE-CLASS* which initially has no instances of its own. However,

insert-object in *PASSENGERS* **with** *p* ← *john-smith*

adds a new token to the extension of *PASSENGERS* with "*p*" *p*-value the person *john-smith*, and returns that new token as value. A token *x* can be removed from the extension of a class *C* through the expression

remove-object *x* **from** *C*

Note that when a token is added to the extension of a class, it is also added to the extensions of all its generalizations, and when it is removed from a class, it is removed from the extensions of all its specializations. Thus Postulate III for the *IS-A* relationship is never violated as a result of an insertion or removal of a token.

In addition to **insert-object** and **remove-object**, TAXIS provides three other QUEL-like ([7]) expressions which allow general searches of the extension of one or more variable classes. Thus the expression

for *x* **in** *EMPLOYEE*
  for *y* **in** *MANAGER*
    retrieve into *FATCATS* **with** *name* ← *x.name*, *sal* ← *x.sal*
      where *x.depth* = *y.dept* **and** *x.sal* > *y.sal*

retrieves into the variable class *FATCATS* employees making more than one of their managers. Note that the assumption (*MANAGER IS-A EMPLOYEE*) implies that *MANAGER* has the properties of *EMPLOYEE*, in particular, *sal* and *dept*.

In addition to **retrieve, append** and **delete** expressions are also provided and are similar in form and semantics to **retrieve** (or corresponding QUEL commands).

Variable classes are the only classes which are allowed to have key properties. Going from a key to the corresponding token is handled in terms of the mechanisms already introduced. Thus if *address*-1 is a particular address,

('*SMITH, JOHN, B*', *address*-1).*person-id*

returns either the person identified by this key or **nothing**.

The attribute factual properties of a variable class instance can be changed through the *update operator* "←". For instance,

john-smith.age ← 35

changes *john-smith*'s *age* from whatever it was to 35.

4.2 Aggregate Classes

A second important category of classes consists of instances of the built-in metaclass *AGGREGATE-CLASS*. The extension of an aggregate class is determined at all times by the cross product of the extensions of its *p*-values. For example, the extension of the aggregate class [| *city*: *CITY*, *country*: *COUNTRY* |] is the cross product of the extensions of *CITY* and *COUNTRY*. The only way to change the extension of an aggregate class is to change the extension of one of its *p*-values.

Instances of aggregate classes can be referenced but never created or destroyed. Thus

[*city*: '*TORONTO*', *country*: '*CANADA*']

references a tuple which is an instance of any aggregate class whose extension includes the tuple ('*TORONTO*', '*CANADA*'). We call the tokens referenced through the matchfix operators [,] *aggregates*.

All the simple properties of an aggregate class are characteristic properties and cannot be changed for any one aggregate. However, there is an expression in TAXIS which allows the identification of an aggregate related to a given one with respect to some of its components. For example, if x is the aggregate [('*TORONTO*', '*CANADA*'] then the expression

x **but** *city* ← '*MONTREAL*'

identifies the tuple obtained from x by replacing its *city p*-value with '*MONTREAL*'.

4.3 Finitely Defined Classes

Instances of the built-in metaclass *FINITELY-DEFINED-CLASS* have their extensions specified once and for all at the time they are defined, e.g.,

CANADIAN-METROPOLES := {| '*MONTREAL*', '*TORONTO*', '*VANCOUVER*' |}

or

INTERNATIONAL-FLIGHT# := {|500 :: 999|} **is-a** *FLIGHT#*

Finitely defined classes are very similar to Pascal scalar types. For instance, the functions *succ* and *pred* return the successor or predecessor of an instance in the ordering of instances specified by the class definition. Similarly, there are special relations *lt*, *gt*, *le*, *ge* which compare two instances of a finitely defined class with respect to this ordering.

4.4 Test-Defined Classes

Aggregate, finitely defined, and formatted classes are all special cases of the general collection of *test-defined classes*. Such classes are characterized by the fact that membership in their extension is determined by a transaction defined for this purpose:

⟨(*ANY*, *TEST-DEFINED-CLASS*), *test*, *TEST-TRANSACTION*⟩

This complex property specializes for aggregate classes to

⟨(*ANY-AGGREGATE*, *AGGREGATE-CLASS*), *test*, *TEST-AGGREGATE*⟩

where *AGGREGATE* is a specialization of *ANY* with all possible aggregates as instances. Similarly, we have

⟨(*ANY-FINITELY-DEFINED*, *FINITELY-DEFINED-CLASS*), *test*, *FINITE-TEST*)⟩

and

⟨(*STRING*, *FORMATTED-CLASS*), *test*, *FORMAT-TEST*⟩

where *STRING*'s extension contains all strings and *TEST-AGGREGATE*, *FINITE-TEST*, and *FORMAT-TEST* are all specializations of *TEST-TRANSACTION*. The essence of these three transactions was already given in the discussion of aggregate, finitely defined, and formatted classes. For instance, *TEST-AGGREGATE*(x, C) checks that the components of aggregate x are instances of the p-values of C's attribute properties. *FINITE-TEST*(x,C), on the other hand, checks whether x is one of the tokens defined to be in the extension of C. Generally, if C is a test-defined class, then

x **instance-of** $C \equiv$ (*Type*(x), *Type* (C)) .. *test* (x,C)

Not all test transactions are predetermined as they are for aggregate, finitely defined, and formatted classes. For example, we can define the metaclass

metaclass *TRAVELER-TO-CANADA-CLASS* **is-a** *TEST-DEFINED-CLASS*

and then the transaction

```
TRANSACTION-CLASS TEST-TRAVELER-TO-CANADA is-a TEST-
  TRANSACTION with parameter-list
    test:(p, class);
  locals
    p: PERSON;
    class: TRAVELER-TO-CANADA-CLASS;
  returns
    rtrn: not (nothing =
        get-object x from RESERVATION
          where (x.person = p and
                    x.flight.destination.country = 'CANADA'))
end
```

thereby setting up the definitional property

⟨(*PERSON*, *TRAVELER-TO-CANADA-CLASS*), *test*,
TEST-TRAVELER-TO-CANADA⟩

Now, the class defined by

TRAVELER-TO-CANADA-CLASS TRAVELER-TO-CANADA **is-a** *PERSON*

has as instances all persons who have booked a reservation for a flight with a destination in Canada.

4.5 Expressions

Expressions can only appear in TAXIS programs as *p*-values of prerequisite, action, result, or return properties.[2]

Conditional, block, and looping constructs are provided in the language for the construction of compound expressions from simpler ones.

Expressions are classes and can have definitional properties of their own (which associate exceptions with them). However, expressions are special types of classes in two respects:

(1) their extension is invariably empty;
(2) their *IS-A* hierarchy is determined by the following rule: If $\langle T,p,E\rangle$ and $\langle T',p,E'\rangle$ and (T *IS-A* T'), then (E *IS-A* E'), where T, T' are transactions, and E, E' are expressions.

Thus there is no need to specify explicitly the *IS-A* hierarchy of expression classes since that is determined by the transactions to which they are attached.

The fact that expression classes have empty extensions means that Postulate III (the extensional *IS-A* constraint) is trivially satisfied for expressions. As a replacement we propose the following postulate.

III′ (Behavioral *IS-A* Constraint) (a) If E, E' are Boolean expressions and (E *IS-A* E'), then it must be that $E \rightarrow E'$ (E implies E') and E causes at least the side effects of E'.

(b) If E, E' are non-Boolean expressions and (E *IS-A* E'), then it must be that when *value* (E) $\neq$ **nothing**, *value* (E) = *value* (E') and moreover E causes at least the side effects of E'.

Consider, for example, a specialization of the *RESERVE-SEAT* transaction, say T, for which the prerequisite *seats-left?* must be redefined. It makes sense, according to the Postulate III′ (the behavioral *IS-A* constraint), to redefine it as

prereq *seats-left?* **on** T **is** *f.seats-left* > 10,

since (*f.seats-left* > 10) → (*f.seats-left* > 0). The redefinition, however,

prereq *seats-left?* **on** T **is** *f.seats-left* > 0 **or** *p.age* < 2

is inappropriate because

(*f.seats-left* > 0 **or** *p.age* < 2) $\not\rightarrow$ *f.seats-left* > 0)

Similarly, the block expression E defined by

```
begin
  insert-object in RESERVATIONS with
    person ← p, flight ← f;
  insert-object in PASSENGERS with p ← p;
end
```

[2] This discussion does not apply to expressions involving @, [| , |], and { | , | } which define new classes and are evaluated at compilation time.

can be made a specialization of *RESERVE-SEAT*.. *make-reservation* because its side effects, which involve two insertions, include those of *RESERVE-SEAT*.. *make-reservation*. The same statement is not true if the first **insert-object** expression is deleted from *E*.

Postulate III′ (the behavioral *IS-A* constraint) is formalized in [25] and its consequences are discussed.

4.6 Transactions

We have already presented the basic categories of properties one can associate with a transaction. Through prerequisites, actions, and results, the TAXIS user can "factor out" a transaction body into semi-independent constraint checks and actions that may be associated with a transaction directly, during its definition, or indirectly, through inheritance.

4.7 Exceptions

We have adapted Wasserman's [21] procedure-oriented exception-handling mechanism with modifications that allow exceptions and exception-handling to be treated within the framework of classes, properties, and the *IS-A* relationship.

Exception classes are defined and organized into an *IS-A* hierarchy, like all other classes. The built-in metaclass *EXCEPTION-CLASS* has as instances all exception classes which are also specializations of the built-in class *ANY-EXCEPTION*. For a particular TAXIS program, or a collection thereof, we may have below *ANY-EXCEPTION* the classes *SECURITY-EXCEPTION*, *CONSTRAINT-EXCEPTION*, etc. Below these, one may wish to attach exception classes such as

```
EXCEPTION-CLASS NO-SEATS-LEFT is-a CONSTRAINT-EXCEPTION with
  attribute-properties
    pers: PERSON;
    flt: FLIGHT;
end
```

When an instance of this exception class is created (i.e., is *raised*), its factual properties are assigned *p*-values through which one can obtain information about the circumstances under which the exception was raised.

Exceptions are raised when a prerequisite or result expression evaluates to a value other than **true**. To specify which exception is raised, one must associate with a prerequisite or result *p*-value, which is always an expression class, an exception class. For *RESERVE-SEAT*, for example, this can be done either by replacing the *seats-left?* property of the transaction with

```
TRANSACTION-CLASS RESERVE-SEAT with
      ...
  seats-left?: f.seats-left > 0 exc
     NO-SEATS-LEFT (pers: p, flt: f);
      ...
end
```

or by adding a definitional property to the *p*-value of the *seats-left?* property with

```
exception-property exc on RESERVE-SEAT.. seats-left? is NO-SEATS-LEFT (pers:
  p, flt: f)
```

In both cases, the associations *pers*: *p*, *flt*: *f* indicate the *p*-values to be assigned to the factual properties of the *NO-SEAT-LEFT* instance raised when the prerequisite *seats-left*? fails.

When an exception is raised within a transaction *T*, it is up to the caller of *T* to specify what should be done to handle it. Such specifications come in the form of complex properties called *exception-handlers* that take as subjects an expression *E* and an exception *EXC* and *p*-value an exception-handling transaction T_h. When an instance of *EXC* is raised during the evaluation of *E*, then T_h is called with the exception raised as its only argument. Suppose, for example, that the transaction *CALLER* calls *RESERVE-SEAT* or one of its specializations during the execution of one of its actions, say *act*. To indicate that the transaction *FIND-ALTERNATIVE* should be called if the exception *NO-SEATS-LEFT* is raised, we write

```
TRANSACTION-CLASS CALLER with
    ...
  actions
      ...
    act: RESERVE-SEAT(p1, f1)
          exc-handler eh for NO-SEATS-LEFT is
            FIND-ALTERNATIVE
      ...
end
```

which defines the complex property

⟨(*RESERVE-SEAT*(*p*1, *f*1), *NO-SEAT-LEFT*), *eh*, *FIND-ALTERNATIVE*⟩

Now, if an instance of *NO-SEATS-LEFT* is raised during the evaluation of *RESERVE-SEAT* (*p*1, *f*1), *FIND-ALTERNATIVE* will be called with the newly created exception instance as argument. From the properties of this instance, *FIND-ALTERNATIVE* will determine the circumstances of the exception and, we hope, what should be done.

Treating exceptions and exception-handling in terms of classes, properties, and the *IS-A* relationship means that the already existing *IS-A* hierarchy of data classes and transactions can be used to structure exception-handling within any one TAXIS program. We illustrate this point by extending the example we have used so far so that if a *NO-SEATS-LEFT* instance is raised for a child, it is not only for the child that an alternative is found but also for his or her guardian. First, we create a specialization of *NO-SEATS-LEFT*:

```
EXCEPTION-CLASS NO-SEAT-FOR-CHILD is-a NO-SEATS-LEFT with
  attribute-properties
    guardian: ADULT;
end
```

Then we redefine the exception property *exc* of the *seats-left*? prerequisite for the transaction (*CHILD*, *INTERNATIONAL-FLIGHT*) . . *reserve-seat*

```
exception-property exc on (CHILD, INTERNATIONAL-FLIGHT) ..
    reserve-seat .. seats-left? is
  NO-SEAT-FOR-CHILD (pers: p, flt: f, guardian: p.guardian)
```

Finally, we augment the exception handler *FIND-ALTERNATIVE* for the exception-handling property *eh* of *CALLER .. act* and *NO-SEAT-FOR-CHILD*:

action *find-alternative-for-guardian-too* **on**
  (CALLER .. act, NO-SEAT-FOR-CHILD) .. eh **is**
  /*remove the child's guardian from the flight *flt* and reserve a seat for him or her as well on the alternative flight selected*/

According to this, another action property is added to the (transaction) class specified by the expression *(CALLER .. act, NO-SEAT-FOR-CHILD) .. eh*. *CALLER .. act* evaluates to the expression class *RESERVE-SEAT*($p1, f1$) (see definition of *CALLER*), and *RESERVE-SEAT*($p1, f1$), *NO-SEATS-LEFT* have a complex property *eh* whose *p*-value is the (exception-handling) transaction *FIND-ALTERNATIVE*. It follows then that the expression (*CALLER .. act, NO-SEAT-FOR-CHILD*) .. *eh* evaluates to a specialization of *FIND-ALTERNATIVE* which inherits all the actions of that transaction in addition to the new action defined by the *find-alternative-for guardian-too* action.

We will not present code for the new action defined for the exception-handler of *NO-SEATS-LEFT* exceptions. It is worth noting, however, that the *IS-A* hierarchy of exception-handlers is patterned after that of *PERSON*, *FLIGHT*, and their specializations, along with the transactions that operate on them.

When an exception-handling transaction completes it execution, control returns to the point where the exception was raised and the expression following the prerequisite or result where the exception was raised is evaluated. Thus each prerequisite or result expression *E* can be interpreted as a conditional expression

if *E* **then nil else**. . .

where the blank is filled by the caller of the transaction where *E* appears.

5. CONCLUSIONS

Several other research efforts are related and/or have influenced our work. PLAIN [22] is one of the few examples of a language designed with goals similar to those of TAXIS. The main difference between the two languages is that PLAIN does not use the *IS-A* relationship as a structuring construct for data or procedures. We have adapted PLAIN's exception-handling mechanism, but modified it to make it consistent with the TAXIS framework. Moreover, due to the structure of transactions, we have managed to restrict the kind of situation under which an exception is raised to failure of a prerequisite or a result.

A recent proposal in [13] for the use of type hierarchy is basically identical to the *IS-A* hierarchy described in this paper. Our work seems to differ from Mealy's only in that his is applied to EL1 data structuring mechanisms [23] rather than the design of an application language.

Our *IS-A* hierarchy is also similar to the generalization hierarchy proposed in [20], although we do not use the "unique key" assumption they impose on their hierarchy, nor do we use their notion of image domains which defines a particular implementation of the *IS-A* relationship within a relational database framework. Another difference between *IS-A* and the generalization hierarchy proposed by the Smiths is that it is possible to redefine a property for a specialization of a class in TAXIS (subject to Postulate IV structural *IS-A* constraint), but that is not the case for the generalization hierarchy. We consider this ability to redefine properties (by specializing their p-values) an important component of the structuring mechanism offered by the *IS-A* relationship. Hammer and McLeod [6] and Lee and Gerritzen [9] have also proposed data models which offer an *IS-A* relationship.

The treatment of the *INSTANCE-OF* relationship in TAXIS is based on the treatment this relationship receives in PSN (procedural semantic network formalism) described in [10, 11]. However, PSN allows an arbitrary number of metaclass levels, as well as the possibility for a class to be an *INSTANCE-OF* itself. We have avoided such a scheme because experience has taught us that two levels of classes are sufficient for most situations. Lee [8] and Smith and Smith [19] also offer proposals concerning the *INSTANCE-OF* relationship.

The high-level relational database operations of QUEL (e.g., **retrieve** [7]) are very similar to the compound expressions used to manipulate variable classes. Obviously, variable classes share many features with relations of the relational model. In embedding variable classes in a programming language we have taken a very different approach from that described in [17] which treats relations as data objects that can be created dynamically as results of relational operations. Instead, in TAXIS *no classes* (variable or otherwise) can be created as results of run-time operations. We rejected Schmidt's proposal very early in our work because it raises a design dilemma for which we do not have a good solution: either we allow the inclusion of classes in TAXIS programs that do not have the usual TAXIS semantics (i.e., properties and a position on the *IS-A* hierarchy), contrary to design principle (2) of Section 1, or we include run-time facilities for obtaining the TAXIS semantics for derived classes, as done in [15], contrary to design principle (4).

Finally, Abrial's work [1] has been very influencial in directing us toward "data models" or "representation schemes" [26] which offer procedural as well as data-oriented facilities for the definition of a model.

From an AI point of view, our work is a direct descendant of PSN, with much of the power of the formalism left out to accommodate the design principles of TAXIS.

As far as contributions are concerned, we believe that this paper has provided evidence on how a framework involving classes, properties (of classes), the *IS-A* relationship, and to a lesser extent the *INSTANCE-OF* relationship, can be used to account not only for data-oriented (declarative, to use the terminology in [26]) aspects of a model of some enterprise, but also procedural ones, e.g., expressions, exceptions, and transactions.

Acceptance of the TAXIS framework for the design of IISs can have far-reaching consequences:

(1) It provides a methodology for dealing with semantic integrity constraints, which in TAXIS are treated as prerequisite and result properties of transactions and are organized into an *IS-A* hierarchy consistent with those defined for data classes and operations on them.

(2) It provides a general design methodology based on "stepwise refinement by specialization" as opposed to "stepwise refinement by decomposition" [24], which has been the main design tool used so far in program development. For

data structures, an account of what stepwise refinement by specialization means and how it relates to stepwise refinement by decomposition has already been given in [20]. TAXIS proposes a similar framework for *all* aspects concerning a program design, not just its data structures. Further evidence for the importance of this notion is provided in [25].

There are four directions along which research on TAXIS is proceeding:

(1) Formalization. TAXIS offers some unusual constructs and a formal definition of what they mean appears highly desirable. Wong [25] provides an axiomatization of the language as well as a denotational semantics to account for these constructs. A by-product of this work is the ability to prove TAXIS programs correct with respect to some logical specification.

(2) Definition of Input/Output Facilities. TAXIS does not offer input/output facilities at this time. To extend it in order to have it provide such facilities, we are considering the possibility of using the same framework (classes et al.) for the definition of all syntactic and pragmatic aspects of a user interface.

(3) Implementation. A TAXIS parser and code generator, and possibly an interactive system through which a designer can use TAXIS, is an important step toward testing the language. Also, there are important theoretical problems such as the mapping of variable and transaction classes into relations and procedures, respectively.

(4) Applications. Apart from the design of individual IISs in TAXIS, we wish to explore the possibility of extending TAXIS to make it suitable for the design of IISs from one particular applications area, say, accounting or inventory control.

ACKNOWLEDGMENTS

We would like to thank Teresa Miao for typing this paper.

REFERENCES

1. ABRIAL, J.R. Data semantics. In *Data Management Systems*, J.W. Klimbie and K. L. Koffeman (Eds.), North Holland Pub. Co., Amsterdam, 1974.
2. BRACHMAN, R. On the epistemological status of semantic networks. In *Associative Networks*, N. Findler (Ed.), Academic Press, New York, 1979.
3. CODD, E.F. A relational model for large shared data banks. *Commun. ACM 13*, 6 (June 1970), 377–387.
4. DAHL, O.J., AND HOARE, C.A.R. Hierarchical program structures. In *Structured Programming*, O.J. Dahl, E. Dijkstra, and C.A.R. Hoare (Eds.), Academic Press, New York, 1972.
5. DATE, C.J. An architecture for high level language database extension. *Proc. 1975 ACM SIGMOD Conf.*, pp. 101–122.
6. HAMMER, M., AND MCLEOD, D. The semantic data model: A modeling mechanism for database applications. *Proc. 1978 ACM SIGMOD Conf.*, pp. 26–36.
7. HELD, G., STONEBRAKER, M., AND WONG, E. INGRES: A relational data base system. *Proc. Nat. Computer Conf.*, Anaheim, Calif., 1975, pp. 19–22.
8. LEE, R., On the semantics of instance in database modeling. Working Paper, Dep. Decision Sci., Wharton School, Univ. Pennsylvania, Philadelphia, 1978.
9. LEE, R., AND GERRITZEN, R. A hybrid representation for database semantics. Tech. Rep. 78-01-01, Dep. Decision Sci., Wharton School, Univ. Pennsylvania, Philadelphia, 1978.
10. LEVESQUE, H. A procedural approach to semantic networks. M.Sc. thesis (Tech. Rep. 105), Dep. Computer Sci., Univ. Toronto, Toronto, Canada, 1977.
11. LEVESQUE, H., AND MYLOPOULOS, J. A procedural semantics for semantic networks. In *Associative Networks*, N. Findler (Ed.), Academic Press, New York, 1979.

12. LISKOV, B., SNYDER, A., ATKINSON, R., AND SCHAFFERT, C. Abstraction mechanisms in CLU. *Commun. ACM 20*, 8 (Aug. 1977), 564–576.
13. MEALY, G. Notions. In *Current Trends in Programming Methodology*, vol. 2, R. Yeh (Ed.), Prentice-Hall, Englewood Cliffs, N.J., 1977.
14. MYLOPOULOS, J., BERNSTEIN, P., WONG, H.K.T. A preliminary specification of TAXIS: A language for interactive systems design. Tech. Rep. CCA-78-02, Computer Corp. of America, 1978.
15. ROUSSOPOULOS, N. A semantic network model of databases. Ph.D. dissertation (Tech. Rep. 104), Dep. Computer Sci., Univ. Toronto, Toronto, Canada, 1976.
16. ROWE, L.A., AND SHOENS, K.A. Data abstraction, views and updates in RIGEL. *Proc. 1979 ACM SIGMOD Conf.*
17. SCHMIDT, J.W. Some high level language constructs for data of type relation. *ACM Trans. Database Syst. 2*, 3 (Sept. 1977), 247–261.
18. SHAW, M., WULF, W.A., AND LONDON, R.L. Abstraction and verification in ALPHARD: Defining and specifying iteration and generators. *Commun. ACM 20*, 8 (Aug. 1977), 553–563.
19. SMITH, J., AND SMITH, D.C.P. A database approach to software specification. Tech. Rep. CCA-79-17, Computer Corp. of America, 1979.
20. SMITH, J., AND SMITH, D.C.P. Database abstractions: Aggregation and generalization. *ACM Trans. Database Syst. 2*, 2 (June 1977), 105–133.
21. WASSERMAN, A.I. Procedure-oriented exception-handling. Tech. Rep. 27, Lab. Medical Inf. Sci., Univ. California, San Francisco, 1977.
22. WASSERMAN, A.I., SHERNIZ, D.D., AND HANDA, E.F. Report on the programming language PLAIN. Lab. Medical Inf. Sci., Univ. California, San Francisco, 1978.
23. WEGBREIT, B. The treatment of data-types in EL1. *Commun. ACM 17*, 5 (May 1974), 251-264.
24. WIRTH, N. Program development by step-wise refinement. *Commun. ACM 14*, 4 (April 1971), 221–227.
25. WONG, H.K.T. Design and verification of interactive information systems. Ph.D. dissertation, Dep. Computer. Sci., Univ. Toronto, Toronto, Canada. To appear.
26. WONG, H.K.T., AND MYLOPOULOS, J. Two views of data semantics: Data models in artificial intelligence and database management. *INFOR 15*, 3 (Oct. 1977), 344-382.

Received April 1978; revised November 1979; accepted December 1979