Galileo: A Strongly-Typed, Interactive Conceptual Language

ANTONIO ALBANO
Universita' di Pisa
LUCA CARDELLI
AT&T Bell Laboratories
AND
RENZO ORSINI
Universita' di Pisa

Galileo, a programming language for database applications, is presented. Galileo is a strongly-typed, interactive programming language designed specifically to support semantic data model features (classification, aggregation, and specialization), as well as the abstraction mechanisms of modern programming languages (types, abstract types, and modularization). The main contributions of Galileo are (a) a flexible type system to model database structure and semantic integrity constraints; (b) the inclusion of type hierarchies to support the specialization abstraction mechanisms of semantic data models; (c) a modularization mechanism to structure data and operations into interrelated units (d) the integration of abstraction mechanisms into an expression-based language that allows interactive use of the database without resorting to a new stand-alone query language.

Galileo will be used in the immediate future as a tool for database design and, in the long term, as a high-level interface for DBMSs.

Categories and Subject Descriptors: D.3.3 [**Programming Languages**]: Language Constructs—*abstract data types; data types and structures;* H.2.1 [**Database Management**]: Logical Design—*data models; schema and subschema;* H.2.3 [**Database Management**]: Languages—*data description languages (DDL); data manipulation languages (DML); query languages*

General Terms: Design, Languages

Additional Key Words and Phrases: Type hierarchy, database semantics, integrity constraints, exception handling.

1. INTRODUCTION

1.1 Motivation

If complex applications utilizing DBMS technology are to be developed, the crucial aspects of these applications must be designed in a high-level language with features that differ considerably from those supported by traditional DBMSs

This work was supported in part by the CNR (Italian National Research Council), Progetto Finalizzato Informatica, Obiettivo DATAID, and in part by the Ministero della Pubblica Istruzione.
Authors' addresses: A. Albano, R. Orsini: Dipartimento di Informatica, Universita' di Pisa, Corso Italia 40, I-56100 Pisa, Italy; L. Cardelli: AT&T Bell Laboratories, Murray Hill, NJ 07974.

[18, 34, 35, 38, 54]. Let us call this activity *conceptual modeling*, its result a *conceptual schema*, and the language used the *conceptual language*.

There are many opinions as to the role of the conceptual schema during the design process. These ideas, which are reflected in the features of the conceptual language, are briefly outlined below.

(1) The conceptual schema documents the database structure in terms very similar to those employed by users to describe an application. It is therefore a model that will be used during the entire life cycle of the database for considerations on the logical data structure and to verify informally that this structure can adequately satisfy user requirements, before the implementation begins. As the conceptual schema is only used for documentation, it does not have to be given in an executable language. The schema is automatically processed only to provide useful reports. An early significant example of this approach is the PSL/PSA design environment [50].

(2) A second class of proposals extends the previous approach in a direction that more closely resembles the software specification problem. This perspective is particularly interesting because of the reciprocal influences of techniques and methodologies [1, 19, 21, 48]. The rationale behind this approach is that since a complex database implementation is a long-term evolving activity, it is essential that the conceptual schema be carefully designed and tested to reduce logical errors in the implementation, and to safely incorporate the new requirements which will arise during the operational phase. The features of the conceptual language are at present still under discussion. In particular, attention is being given to operational aspects, besides the structure of data. However, in addition to abstract specifications, there are a number of pragmatic reasons why a high-level, executable language should be used in conceptual design. In fact, such a language could also be used to test the adequacy of the conceptual schema, if we do not care about execution efficiency [16, 26, 34, 39].

(3) A third class of proposals considers a conceptual language as a tool that is much more than a mere step on the way to implementing a database application. To design complex, interactive computerized information systems, a programming language with abstraction mechanisms to model databases is needed.

There are a few proposals that have adopted the last approach, although general agreement has been reached only with regard to certain basic features of the language [22, 31, 32, 37, 49]. Such a language should provide constructs to aid the designer in expressing, as far as possible, the semantics of the application in the conceptual schema, rather than in the application programs. At the least it should provide the following features:

(a) data defined both declaratively, with abstraction mechanisms (aggregation, classification, and specialization), and procedurally (derived data);
(b) semantic integrity constraints, both standard (such as keys and mandatory values) and those described by a general-purpose constraint specification language;
(c) operations to give the behavioral semantics of the data in the schema; and
(d) a sound mathematical foundation for the language.

1.2 Assumptions

This paper describes the features of the conceptual language Galileo, a programming language that supports semantic data model features. It therefore belongs to the third approach outlined above. The features presented here have been partially implemented [4]. These are the first results of a project that aims at designing and implementing a prototype system, Dialogo, to experiment with a stand-alone programming environment and to support the development and testing of database design [2].

The approach adopted in designing Galileo takes into account the requirements previously described, but also assumes that a conceptual language should provide

(a) a set of independent constructs to be used in any combination to achieve simplicity and expressiveness;
(b) features to design and test the solution incrementally;
(c) a modularization mechanism to decompose the design into meaningful modular units that correspond to a description of the database at different levels of successive refinements, or to application-oriented views of the database.

1.3 Relation to Previous Work

The design of Galileo has been influenced by two areas of research: conceptual modeling and programming languages. Although these areas have a number of overlapping issues, there are problems to be solved if the results from both areas are to be successfully integrated [18, 20].

Galileo applies results from the conceptual modeling sector for features related to object-oriented databases, declarative definitions of constraints, multiple descriptions of objects, and view modeling [11, 13, 17, 31–33, 35, 37, 42, 45, 47, 49, 53].

Galileo borrows features such as data types, abstract types, and modularization from the programming language area [44]. Although the utility of such features is recognized both pragmatically and theoretically, they have mainly been studied for applications to temporary data (i.e., not involving databases).

The database proposals that have most influenced the design of Galileo are TAXIS, DIAL, and ADAPLEX. TAXIS is notable for introducing the basic knowledge representation mechanisms of semantic networks on data, transactions, and exceptions and for its approach to user dialogue modeling [12]. DIAL, which has evolved from SDM [32], uses data types, classes, derived classes, the "port" mechanism to deal with user interaction, and features to control concurrency at the conceptual level [31]. Finally, ADAPLEX uses semantic data model features in a strongly-typed programming language, namely, Ada [49].

The main contributions of Galileo are

(1) the integration of features to support semantic data model abstraction mechanisms within an expression-based, strongly-typed programming language;
(2) a systematic use of both concrete and abstract types to model structural and behavioral aspects of a database;
(3) the inclusion of type hierarchies to support the specialization abstraction mechanism of semantic data models as well as a software development methodology by data specialization [5, 14];

(4) the proposal of another abstraction mechanism, modularization, to organize a conceptual scheme in meaningful and manageable units, so as to deal with data persistence without resorting to specific data types such as files of programming languages, and to deal with application-oriented views of data in a similar way to the view mechanism of DBMSs [7];
(5) a small number of independent primitive features that can be applied orthogonally, that is, in any combination.

The basic ideas of Galileo have been investigated in ELLE, a programming language designed to deal uniformly with temporary and persistent complex data, that is, without resorting to special data type constructors to deal with permanent data [3]. Both ELLE and Galileo borrow many of their features from the functional programming language ML [29]. A comparison of ADAPLEX, DIAL, Galileo, and TAXIS is reported in [15].

1.4 Structure of the Paper

The purpose of this paper is to illustrate the features of Galileo. The schema fragments used as examples are intended only to illustrate the main concepts, and do not cover all the language features. A complete description of Galileo exists as a technical report [6]. The semantics of Galileo are described informally, but its formalization, using a denotational approach, is reported in [24, 41]. Pragmatic aspects of Galileo (schema design methodology and designers' reactions to the language) are also beyond the scope of this paper; they are currently being studied in the context of a joint project by a group of Italian universities and companies sponsored by the Italian National Research Council (CNR) [27]. The goal of the project is the development of a database design methodology, together with a set of integrated computer-assisted tools covering all aspects of the design process, including application analysis, conceptual modeling, and logical and physical design in both centralized and distributed environments.

The next section presents the basic data modeling features of Galileo. Section 3 describes the operators that affect the environment used to evaluate expressions. Section 4 describes the type system of the language, and Section 5 the notion of type hierarchies. Section 6 presents the class mechanism used to build an object-oriented view of a database, with classification, aggregation, and specialization abstraction mechanisms. Section 7 presents the modularization mechanism to structure a schema and to deal with persistent data. Section 8 illustrates the failure-handling mechanism together with transactions modeling. In the conclusions, we comment upon the implementation now underway and on our future plans.

2. OVERVIEW OF GALILEO

Galileo supports the following abstraction mechanisms for database modeling:

Classification. Entities being modeled which share common characteristics are gathered into *classes.* All elements of a class have the same type. The name of the class denotes the elements present in the database. The elements of a class are represented uniquely, that is, only one copy of each element is allowed.

Aggregation. Elements of classes are aggregates; that is, they are abstractions having heterogeneous components, and may have elements of other classes as

components. Associations among entities are represented by aggregations in a Galileo database. Components of aggregates can be collections of homogeneous values to represent, for example, multivalued associations among entities. Because of the unique representation of elements of classes, any modification of an element is reflected everywhere that element appears as a component.

Generalization. Elements of a class can be described in different ways by means of subclasses. Subclasses are derived from classes by using a predefined set of operators. Elements of a subclass also belong to their parent class. The type of the elements of a subclass is a subtype of the type of elements of the parent class. The subclass mechanism includes the IS-A hierarchy of semantic networks and semantic data models.

Modularization. Data and operations can be partitioned into interrelated modules. A complex schema can therefore be structured into smaller units. For instance, a unit may model a user view or a description of the schema produced by a stepwise refinement methodology.

Galileo also has the following features:

(1) It is an expression language; each construct is applied to values to return a value.
(2) It is an interactive language; the system repeatedly prompts for inputs and reports the results of computations; this interaction is said to happen at the top level of evaluation. At the top level one can evaluate expressions or perform declarations. This feature allows the interactive use of Galileo without a separate query language.
(3) It is higher order, in that functions are denotable values of the language. Therefore, a function can be embedded in data structures, passed as a parameter, and returned as a value.
(4) Every denotable value of the language possesses a type:
    (a) A type is a set of values sharing common characteristics, together with the primitive operators which can be applied to these values.
    (b) The predefined types of the language are **bool**, **num**, **string**, equipped with the usual operators, and the type **null**, which is a singleton set with the element **nil** equipped with the equality operator.
    (c) The type constructors available to define new types, from predefined or previously defined types, are tuple (record), sequence, discriminated union (variant), function, modifiable value (reference), and abstract types. There are two constructors for abstract types: $\Leftrightarrow$ and $\leftrightarrow$. The former is similar to CLU clusters, ALPHARD forms, or Euclid modules: it is used to define a new type together with the available operations. The latter is similar to the type constructor of Ada: it defines a new type which inherits the primitive operations of the representation type.
    (d) The type system supports the notion of type hierarchy; if a type t is a subtype of a type t', then a value of t can be used as argument of any operation defined for values of t', but not vice versa, because the subtype relation is a partial order.
(5) Every Galileo expression has a type. The meaning of "an expression e having type t'' is that the value of e possesses the type t. In general, any expression has a type that can be statically determined, so that every type violation can

be detected by textual inspection (static type checking). However, if the type checker is not able to ascribe a type to an expression, the user must specify the type with the notation "Expression: Type". The language has been designed to be statically type-checkable for two reasons: first, for the considerable benefits in testing and debugging; second, because programs can be safely executed disregarding any information about types at run time. Execution time testing will be required for constraints only. Finally, static type checking allows a typechecker to give the correct meaning to overloaded operators (i.e., operators that can be used with operands of different types).

(6) Class elements possess an abstract type and are the only values which can be destroyed. Predefined assertions on classes are provided and, if not otherwise specified, the operators for including or eliminating elements of a class are automatically defined.

(7) A control structure is provided for failures and their handling.

The following simple schema illustrates Galileo. The example concerns departments and employees in a firm. The definitions are collected in the Organization schema.

```
Organization := (
  rec Departments class
    Department ↔
      (Name: string
      and Budget: num
      and Address: Address
      and Manager: var Employee
      and Employees: var seq Employee)
      key (Name)

  and Employees class
    Employee ↔
      (Name: string
      and Salary: var num
      and NameOfDept :=
        derived Name of
          get Departments with this isin (at Employees))
      key (Name)

  and NewEmployee(AName: string, ASalary: num): Employee :=
    mkEmployee (Name := AName and Salary := var ASalary)

  and VipEmployees subset of Employees class
    VipEmployee ↔
      (is Employee
      and VipProperty: string)

  and type Address :=
    (Street: string
    and Zip: string
    and City: string)

  drop mkEmployee)
```

The **rec** is used for recursive functions or for mutually dependent types, such as Department and Employee.

Departments and Employees are examples of base classes, while **key** is an example of predefined class constraint to assert that the elements of the classes must differ in the value of the Name attribute.

An attribute of an element of a class may be *primitive* or *derived*. A primitive attribute is one that is subject to direct initialization and updating. The value of a derived attribute is automatically computed from other information in the database and cannot be updated: every time the value of the attribute is used, it is as if the associated expression were evaluated to derive the value. An example of a derived attribute is NameOfDept in Employees, where "this" is bound to the current element of the class.

An attribute can be modified if and only if it is defined as type **var**, otherwise it is constant, and any attempt to update the value is detected statically.

The function NewEmployee is an example of a defined operation included in the schema. It is the only operation that can be used to create new elements of the class Employees, since the **drop** operator prevents the predefined mk-Employee operation from being exported outside the schema definition. For Departments and VipEmployees, the functions mkDepartment and mkVip-Employee are available.

VipEmployees is an example of a subclass. It contains all those employees who are believed to be very important. The elements of a subclass must have a type that is a subtype of the elements of the parent class. For instance, the type of the elements of VipEmployees is that of Employee with the additional attribute VipProperty.

This example shows how classes are used to deal with sets of related objects. The approach has some similarity to that adopted for relational databases: in both cases the associations among data are described by means of the value of an attribute. However, in relational databases, data are tuples of simple values collected in relations, and associations among them are represented by assigning as value to an attribute the key value of another tuple. To represent associations in Galileo, the mechanism of "data sharing" is used instead, so that an element of a class can be shared as a component by many others.

3. THE BASIC ENVIRONMENT OPERATORS

An important notion in Galileo is that of *environment*, as it used in the denotational semantics description of programming languages [51]. It is useful to distinguish between the definition of an environment and its run-time interpretation.

An *environment definition* is a map from identifiers to definitions of types or values; it is used to typecheck declarations and expressions before their evaluation.

A *run-time environment* is a map from identifiers to denotable values of the language, obtained by evaluating an environment expression. The evaluation of any expression takes place in the context of an environment, which specifies what the identifiers in use denote. Types are not present in run-time environments since they are not denotable values; that is, types cannot be produced as the result of expressions.

An environment definition is given with the following operators, where A and B stand for environment expressions.

Id := Term	Introduces a new binding between the identifier Id and Term, which is the definition of a value or a type.
A **and** B	introduces the bindings of A and B, but the bindings of A cannot be used in B and vice versa.
A **ext** B	introduces the bindings of B and those of A not redefined in B. The bindings of A can be used in B, but not vice versa. In other words, A is extended with B.
rec A	introduces the bindings of A which can be used recursively in A.
type A	introduces the bindings between identifiers and types defined in A.
A **drop** Id	introduces the bindings of A, except the one with binder Id.
A **take** Id	introduces only the binding with binder Id defined in A.
A **rename** Id **by** NewId	introduces the bindings of A, but the binder Id is renamed as NewId.

For instance

```
type b := int
ext rec fact(x:b):b :=
  if x = 0 then 1 else x*fact(x − 1)
ext a := fact(3)
ext c := fact(4)
```

The binders defined are *b*, *fact*, *a*, and *c* bound respectively to the type int, the factorial function, the expression fact(3), and the expression fact(4). Once this environment expression has been evaluated, it denotes the set of associations (a, 6), (c, 24), and (fact, the internal representation of the function).

The expression "**use** A **in** Expression" evaluates "Expression" in the current environment temporarily extended with the bindings of A.

```
use a := 3
and b := 4
in a + b      yields 7
```

Other environment operators will be introduced in the sequel to this paper.

4. THE TYPE SYSTEM

All denotable values of the language possess a type. A *type* is a set of values, possibly infinite, together with the primitive operations that can be applied to these values. The predefined types of the language are **bool**, **num**, and **string**, equipped with the usual operators, and **null**, which is a singleton set whose only element is **nil**, equipped with the equality operator.

Type constructors exist to define a type for the following values: tuples, discriminated unions, sequences, modifiable values, functions, and abstract values.

4.1 Tuples

The data structure tuple, such as the records of programming languages and traditional database models, consists of a set of ⟨identifier (attribute or label), denotable value⟩ pairs. The order of the pairs is unimportant. Examples of denotations of tuples are

```
PaulBrown :=
  (Name := "Paul"
  and Surname := "Brown"
  and BirthDate := "06/12/1941")
Department :=


  (Name := "Computer Science"
  and NumOfEmployee := 10
  and Chairman :=
    (Name := "John"
    and Surname := "Moore"
    and Salary := 80))
```

We say that a value is associated with an identifier when it appears in a pair together with that identifier.

Tuples are equipped with the **of** operator which returns the value associated with an identifier (**of** is right associative):

```
Name of
  (Surname := "Moore"
  and Name := "John"
  and Salary := 80)      yields "John".
```

A tuple type consists of an unordered set of pairs ⟨identifiers, type⟩. Two tuple types are equal if they have equal sets of pairs.

Tuples in Galileo are just environments constructed with any environment operators except **type**, although we continue to use the two terms to indicate their use as a data structure (tuple) or as a binding in which evaluation takes place (environment). The following example shows how to construct and use circular data with the operators **rec**, **and**, and **use**:

```
use rec Cs :=
  (Name := "Computer Science"
  and Budget := 100
  and Chairman := Smith)
and Smith :=
  (Name := "John"
  and Salary := 100
  ext Deductions := Salary* 0.1
  and Department := Cs)
in
  Deductions of Chairman of Cs      yields 10
```

A discriminated union, or variant, type consists of a set of alternative values. It is different from the mathematical union of sets in that each value retains an inspectable *tag*, indicating the alternative to which it belongs. Two variant types are equal if the sets of their pairs ⟨tag, type⟩ are equal. An example of variant

type is

type Employee :=
⟨Technician: (Name: string **and** Skill: string)
or Secretary: (Name: string **and** TypingSpeed: string)⟩

Values of such a type are denoted by giving the expected tag:

JohnSmith := ⟨Secretary := (Name := "John Smith" **and** TypingSpeed := "High")⟩
MarySmith := ⟨Technician := (Name := "Mary Smith" **and** Skill := "Analyst")⟩

Two basic operators are defined on variants: **is**, to test the tag of a variant value, and **as**, to get the value contained in the variant. Suppose w denotes a value of type Employee, then a legal Galileo expression is

if w **is** Technician
then Skill **of** (w **as** Technician)
else TypingSpeed **of** (w **as** Secretary)

The case construct is a convenient form to test the tag of a variant and to bind the value to a local identifier:

case w **when**
⟨Technician := x. Skill **of** x
or Secretary := y. TypingSpeed **of** y⟩

The Pascal-like enumeration type ⟨Id **or** . . . **or** Id⟩ is an abbreviation for ⟨Id:null **or** . . . **or** Id:null⟩, and values of such a type can be denoted with ⟨Id⟩ instead of ⟨Id := nil⟩. "**optional** t" is an abbreviation for ⟨bound: t **or** unbound: null⟩. If x is a value of type "**optional** t", it can be used in any expression as an abbreviation for "x **as** bound".

4.2 Sequences

A sequence is a finite ordered collection of homogeneous elements (i.e., data with the same type). Sequences differ from sets in the ordering and multiplicity of elements.

[3; 4; 6*3; 4] is a sequence of integers
[(Name := "Jim" **and** Age := 20);
(Name := "Alice" **and** Age := 31)] is a sequence of tuples

A sequence type is denoted by **seq** followed by the type of the elements. For instance, the following are the types of the above sequences:

seq num
seq (Name: string **and** Age: num)

Since each expression must have a type that is statically determinable, empty sequences must be followed by their types, as in

[]: **seq** num
[]: **seq** (Name: string **and** Age: num)

Two sequences are equal when they meet three conditions: they have the same element types, the same cardinality, and their elements are pairwise equal, in the correct order. Two sequence types are equal if they have equal element types.

The following examples show some operators on sequences:

first [2;3;2]	yields 2
rest [2;3;2]	yields [3;2]
[1;2] **append** [3;4;2]	yields [1;2;3;4;2]
setof [1;2;2;1]	yields [1;2]
3 **isin** [2;3;5]	yields true
emptyseq [2]	yields false

first and **rest** generate a failure when applied to an empty sequence.

all x **in** [2;3;2;3;6] **with** $x > 2$ yields [3;3;6]
all p **in** [(Name := "Jim" **and** Age := 20); (Name := "Alice" **and** Age := 31)]
with Age **of** $p > 20$ yields [(Name := "Alice" **and** Age := 31)]

The following semantically equivalent expression is preferred for sequences of tuples, since it avoids the introduction of the explicit binder:

all [(Name := "Jim" **and** Age := 20); (Name := "Alice" **and** Age := 31)]
with Age > 20

To evaluate an expression for each element of a sequence, such as "Select the names of persons aged more than 20", the following expression can be used:

for [(Name := "Jim" **and** Age := 20);(Name := "Alice" **and** Age := 31)]
with Age > 20 **do** Name yields the sequence ["Alice"]

The conventional aggregate functions sum, average, and so on, are available for sequences of numbers.

4.3 Modifiable Values

Values associated with the previous types cannot be modified. To introduce "modifiability" in the language, for example, to modify the value of a tuple pair or to change the value associated with an identifier in the environment, a new kind of value, the *location*, is introduced. Its name and meaning is one that is commonly used in the denotational semantics description of programming languages [51]. Locations reside in a time-varying structure, the store, and are associated with values of any type, including other locations, since they are also denotable values. The expression "**var** 3" denotes a new location which is associated in the store with the value 3. The type of "**var** Expression" is "**var** TypeOfExpression", and two location types are equal if and only if their associated types are equal.

The operations on locations are getting the associate value, that is, that content of the location; replacing the associated value with a new value of the same type (assigning a value); and testing for equality between locations. For instance,

use x := **var** 3
in at $x + 1$ yields 4

The evaluation of **at** x gives the value associated with the declared location.

The assignment operator ← is an infix binary operator. The value of the left operand must be a location, while the value of the right operand must be a value of the same type as the previous content of the location. This operation modifies

the store replacing the old value of the location and returns **nil**. For example,

use x := **var** 3
in ($x \leftarrow$ **at** $x + 1$; x) yields 4

where ($E1; \ldots; En$) evaluates all expressions Ei sequentially and returns the value of the last one.

4.4 Functions

Functional types are built by the operator $\rightarrow$. The type ($tx \rightarrow ty$) consists of all the functions that map values of type tx to the result of type ty. The expression "**fun**(x: tx): ty **is** Expression" denotes a function with a formal parameter x and a body Expression that returns a value of type ty. The function possesses a type ($tx \rightarrow ty$). To define a function f with formal parameter x and body Expression, one performs the declaration "$f(x: t)$: t' := Expression", equivalent to "f := **fun**(x: t): t' **is** Expression". To apply f to an actual parameter p, one evaluates the expression "$f(p)$". The body of f is evaluated in the environment where f is defined (static scoping) and extended with the bindings (formal parameters, value of the actual parameter). The value of the body is returned as the result of the application. The control structures available to define compound expressions are sequencing, selection, repetition, and failure handling; these will be discussed in Section 8.

4.5 Abstract Types

The types of the values presented so far depend on the structure of the values only. That is, the type compatibility rule adopted is the so-called *structural equivalence* rule. User-defined type names are used as abbreviations for the structures they represent. These types are called *concrete*, in contrast with a new kind of type, called *abstract*. Two user-defined abstract types are always different (i.e., the type compatibility rule adopted for them is the so-called *name equivalence* rule).

Abstract types are not abstract in the sense of algebraic abstract types, but rather are analogous to CLU clusters, ALPHARD forms, and Euclid modules. They are mechanisms to abstract representations of the data from their behavior. Such behavior is defined by the designer in terms of the operations that can manipulate the data. However, an abstract type can be used like any other type in all the contexts where a type is expected. That is, user-defined abstract types have the same status as primitive types, which can be regarded as predefined abstract types provided by the language.

The main reason for introducing abstract types is protection, that is, to provide a mechanism to define a new type together with the operations available on values of that type. Thus, values of different abstract types are not compatible, even though they have the same representation (e.g., a weight is different from a height, although both are represented by integers). In this way, it is possible to tailor unique operations for each type, which cannot be used for objects of other types. For example, a function that tests a height and an age against a table of standards cannot be misused by applying it to a weight and an age. Another

important protection introduced by abstract types is that programs are independent of changes in data representation as long as the primitive operations are the same.

To define abstract types, Galileo offers the following environment operator:

type Id ⇔ Type {**assert** [with "Name"] BoolExpr}

This environment expression introduces the following bindings:

(1) Id is bound to a new type with a domain isomorphic to the domain of the representation type, Type, possibly restricted by the assertions.
(2) The identifiers mkId and repId are bound to two primitive functions, declared automatically, to map values of the representation type into the abstract one and vice versa:

mkId: Type → Id
repId: Id → Type

If an **assert** clause is present, BoolExpr is a Boolean expression on the values of the type. The assertions impose constraints on data values, which are controlled at execution time, when the data is created. If an assertion is violated, the operation fails with the name of the operation or with the name of the assertion, if present.

```
type Time ⇔ (hrs: num and mins: num)
  assert use this
    in hrs within (0,23) And mins within (0,59)
```

This declaration defines an abstract type Time, together with the primitive functions mkTime and repTime. As an abbreviation, constraints on a property can be specified directly in the corresponding pair declaration:

```
type Time ⇔
  (hrs: num this within (0,23)
   and mins: num this within (0,59))
```

To define an abstract type with the representation hidden, but with user-defined operations, the following definition might be used:

```
type Time ⇔
  (hrs: num this within (0,24)
   and mins: num this within (0,60))
with Hours(t: Time): num :=
  hrs of repTime(t)
and Minutes(t: Time): num :=
  mins of repTime(t)
and MakeTime(x: num,y: num): Time :=
  mkTime((hrs := x and mins := y))
```

This declaration exports an abstract type Time, together with three functions MakeTime, Hours, and Minutes. The two primitive functions mkTime and repTime are only available in the definitions that appear in the **with** part, but they are not exported in the scope of the type declaration. The **with** construct is not a special syntax for abstract types, but it is another environment operator: A **with** B means that the types in A can be used in B, and they are exported

together with the definitions in B; the values in A (like mkTime and repTime) can be used in B, but they are not exported. Abstract types are obtained from the interaction of two orthogonal features: the isomorphism constructor ⇔ and the environment operator **with**. Mutually dependent types can be defined with the expression:

```
type rec
  (u ⇔ ···
  and v ⇔ ···
        ···
  and z ⇔ ···)
with op(···) := ···
        ···
 and op(···) := ···
```

To define new types, Galileo provides an additional environment operator:

type Id ↔ Type {**assert** [Name]BoolExpr}

This operator introduces the following bindings:

(1) A new type that *inherits* the primitive operators on the representation type. The primitive operators retain their names, but this overloading does not introduce ambiguities because the typechecker can infer the meaning of an operator from the type of the operands. To restrict the set of operators to be inherited, the operators **drop** or **take** on the representation type might be used.
(2) The identifiers mkId and repId, as for the ⇔ operator.

This environment operator has been included, since, in many cases, most of the primitive operators on the representation type are also needed for the abstract type, especially in database applications. The protection required is that the operators must never be applied to values of different types; and this is the effect of introducing a new type with this operator. When all the operators on the representation type are inherited, this operator is equivalent to the type constructor in Ada, where user-defined types are always different.

```
type PersonAge ↔ num this within (0,150)
  drop mod,*
```

This declaration introduces:

(1) The new type PersonAge with a domain isomorphic to a subset of numbers.
(2) The primitive functions mkPersonAge and repPersonAge.
(3) The predefined operators on numbers translated on the type PersonAge, except mod and *. The operators incorporate the control of the assertion, so the expression "mkPersonAge(10) + mkPersonAge(1)" is equivalent to mkPersonAge(10 + 1).

For example, another definition of Time, which introduces a new type equipped with the selector operators "Hours **of**" and "Minutes **of**" and the functions mkTime and repTime, is

```
type Time ↔
  (Hours: num this within (0,23)
  and Minutes: num this within (0,59))
```

In defining a new tuple type with the operator ↔, it is possible to declare a pair as *default* or *derived:*

```
Type Product
  (Code: string
   and SaleTax: default 0.06
   and Price: var num
   and Cost: var num
   ext Profit:= derived(Price − Cost))
```

This declaration has the following meanings:

(1) In the parameter of the function mkProduct, the derived attributes are ignored, and if default attributes are omitted, the specified value is assumed.
(2) Every time the selector "Profit **of**" is used on a value of type Product, the associated expression is evaluated and its result is returned. If the derived attribute is defined with the **ext** operator, the expression is evaluated extending the definition environment temporarily with the pairs of the tuple. When the **and** operator is used, the function is evaluated in the definition environment.

5. TYPE HIERARCHIES

An important property of the Galileo type system is the notion of subtype: if a type u is a subtype of a type v ($u \subseteq v$), then a value of type u can be used in any context where a value of type v is expected, but not vice versa. The subtype relation is a partial order. For instance, if a function f has a formal parameter of type v, then an application of f to a value of type u is correctly typechecked because no run-time errors can occur. It is important to stress the point that since Galileo has a secure type system, the notion of type hierarchies is related to that of well-typed expressions [24, 28]: expressions that are syntactically well typed are always semantically well typed (i.e., such expressions do not cause run-time type errors, and give a value of the correct type). In Milner's words, "well-typed expressions do not go wrong" also [36] apply to hierarchies among types.

This notion of type hierarchies is different from the subtype concept of Ada, but is similar to the subclass mechanisms of Simula 67 and Smalltalk. In Galileo, this notion is extended to all the types, in the sense explained in the sequel to this paper, while preserving two important properties: the language is still strongly-typed and the functions need not be recompiled in order to be used on parameters of any subtype.

With this mechanism Galileo supports the notion of *programming by data specialization*, originally introduced in Simula 67 and generalized in TAXIS to all the constituents of a database application: data, transactions, assertions, and scripts [14]. Complex software applications, especially those related to databases, can be designed and implemented incrementally. Once a set of functions has been designed and tested for the most general data, it can be used with data of any subtype introduced later on in the software development process. Moreover, new functions on the subtypes can be defined in terms of the old functions.

The subtype relation is automatically inferred by the typechecker for concrete types, but it must be declared explicitly among abstract types. The rules followed

by the typechecker are

(1) For any type t, $t \subseteq t$.
(2) If r and s are tuple types, then $r \subseteq s$ iff:
    (a) the set of identifiers of r *contains* the set of identifiers of s, and
    (b) If r' and s' are the types of a common identifier, then $r' \subseteq s'$.

For instance, if

```
type (Address :=
        (Street: string
        and Zip: string)
      and VipAddress :=
        (Street: string
        and Zip: string
        and Country: string)
      and Person :=
        Name: string
        and Address: Address)
      and Student :=
        (Name: string
        and Address: Address
        and School: string)
      and VipPerson :=
        (Name: string
        and Address: VipAddress))
```

then

Student ⊆ Person
VipPerson ⊆ Person

while it is false that

Person ⊆ VipPerson,
Person ⊆ Student,
Student ⊆ VipPerson and
VipPerson ⊆ Student.

(3) If r and s are variant types, then $r \subseteq s$ iff:
    (a) the set of tags of r is *contained* in the set of tags of s, and
    (b) if r' and s' are the types of a common tag, then $r' \subseteq s'$.

For instance, if

```
type (Day :=
          ⟨Monday or Tuesday
          or Wednesday or Thursday
          or Friday or Saturday
          or Sunday⟩
  and Weekend := ⟨Saturday or Sunday⟩)
then
  Weekend ⊆ Day.
```

(4) If r and s are sequence types with elements of types r' and s', then $r \subseteq s$ iff $r' \subseteq s'$.
(5) A modifiable type "**var** r" is a subtype of another type "**var** s" iff r and s are the same type.

To clarify the reason for this rule, consider the following expression evaluated in an environment containing the previous type definitions.

```
use type Traveler :=
  (Name: string
  and Address: var Address)
ext Agnelli :=
  (Name:= "Gianni Agnelli"
  and Address :=
    var (Street:= "200 Bloor St, Toronto"
      and Zip:= "M4V 2H5"
      and Country:= "Canada"))
and ChangeAddress (x :Traveler, y: Address) :=
  Address of x ← (Street:= Street of y and Zip:= Zip of y);
in
  (ChangeAddress(Agnelli,
    (Street := "New Address"
     and Zip := "New Zip"
     and Country := "New Country"));
Country of at (Address of Agnelli)
```

The application of ChangeAddress is not well typed according to the above rule because the type of Agnelli is not a subtype of Traveler. If, for instance, a different rule had been adopted, say that two types **var** r and **var** s are in the $\subseteq$ relation if $r \subseteq s$, then the previous expression would have been accepted by the typechecker, but it would no longer be true that "well-typed expressions do not go wrong": the last expression will generate a run-time error because the tuple Agnelli has lost the pair with attribute Country! This is a consequence of the assignment operation in the ChangeAddress function: it assigns a new data value of type (Street: string **and** Zip: string) to the Address of the actual parameter.

(6) If $(r \rightarrow s)$ and $(r' \rightarrow s')$ are function types, then $(r \rightarrow s) \subseteq (r' \rightarrow s')$ iff $r' \subseteq r$ and $s \subseteq s'$.

Note the inversion of the subtype relation between the domains of the functions. To clarify the reason for this rule, consider the following expression (a parameter of type $(r \rightarrow s)$ means that the actual parameter can be any function mapping values of type r to values of type s):

```
use type
  (Person := (Name: string)
  and Student :=
    (Name: string
    and School: string)
  and ForeignStudent :=
    (Name: string
    and School: string
    and Country: string))
and John :=
  (Name := "John")
and JohnStudent :=
  (Name := "John"
  and School := "UofT")
and AnItalian:=
  (Name := "Mario"
  and School: "UofT"
```

```
  and Country: "Italy")
and NameOfPerson (x: Person): string :=
  Name of x
and CountryOfForeignStudent (x: ForeignStudent): string :=
  Country of x
and StringFromStudent (g: Student → string, x: Student): string :=
  g(x)
in
  (StringFromStudent (CountryOfForeignStudent,JohnStudent);
  StringFromStudent (NameOfPerson, AnItalian))
```

For the above rule, the first application of StringFromStudent is not well typed because the type of CountryOfForeignStudent (ForeignStudent → string) is not a subtype of (Student → string). In fact, if it were executed, a run-time error would occur because of the use of the selector "Country **of**" in the function CountryOfForeignStudent on a value of type Student. In contrast, the second application of StringFromStudent is instead well typed.

(7) A type Id ↔ t(the same rule applies to ⇔) is a subtype of another type Id′ ↔ t', with primitive types considered as predefined abstract types, when the subtype relation is declared explicitly to the typechecker as follows:

Id **is** Id′ ↔ t''"NewAssertions", and $t \subseteq t'$

Note that the assertions on Id are those of Id′ plus "NewAssertions".

```
type (PersonAddress := (HomeAddress: string)
  and StudentAddress :=
    (HomeAddress: string
    and College: string)
  and Person ↔
    (Name: string
    and Age: num this within (0,150)
    and Address: PersonAddress)
  and Student is Person ↔
    (Name: string
    and Age: num this within (6,25)
    and School: string
    and Address: StudentAddress))
```

The following abbreviation, used when the representation type is a tuple type, makes evident that the subtype Student inherits attributes and assertions of the type Person:

```
type Student ↔
  (is Person
   and School: string
   ext Address: StudentAddress
   assert use this in Age within (6,25)
```

In the abbreviated notation, the **ext** operator must be used to redefine the type of Address. A derived attribute cannot be redefined in a subtype.

Finally, multiple hierarchies are declared as "Id **is** Id′, Id″ ↔ t", where $t \subseteq t'$, and $t \subseteq t''$, or in the abbreviated form "Id ↔ **is** Id′, Id″". Note that in the abbreviated form, if a common identifier is presented with type tr' in t' and tr'' in t'', then tr' must be a subtype of tr'' or vice versa. In the representation of type Id, the identifier will have the most specialized type.

6. CLASSES

Classes provide a mechanism for representing a database by means of sequences of modifiable interrelated objects. An element of a class is an object that is the computer representation of certain facts of an entity of the world that is being modeled. An object-oriented view of a database is characterized by the following [14, 33, 35]:

(1) There is a one-to-one correspondence between objects in the database and entities of the world that is being modeled.
(2) The objects of the database are all distinct, and they might not have an external reference, such as a key, that stands for them.
(3) Associations among entities are modeled by relating the corresponding objects and not the external references. Moreover, only objects that exist in the database can be used to model associations.

A class is characterized by a name and the type of its elements. The name of a class denotes the elements of the class currently present in the database, while the type gives the structure of the elements. The type of the class elements must be abstract, therefore two elements of different classes are always of different types, although they may be defined with the same representation.

Elements of classes are the only values in Galileo that can be created and destroyed. Moreover, they are uniquely represented, and when updated their modification is reflected in all other objects in which they appear as components.

Each class can be either a *base* class or a *subclass*. A base class is defined independently of other classes, while a subclass is defined in terms of other classes. As in SDM [32], a base class is used to model a primitive collection of entities, while a subclass is used to model alternative ways of looking at the same entities.

6.1 Base Classes

A base class is defined by the environment operator **class**, as shown in the following example with two mutually defined classes:

```
rec Departments class
  Department ↔
    (Name: string
     and Budget: num
     and Address: string
     and Manager: optional Employee
     and Employees: var seq Employee)
     key (Name)


and Employees class
  Employee ↔
    (Name: string
     and Salary: num
     and NameOfDept :=
       derived Name of
         get Departments with this isin (at Employees)
     key (Name)
```

The **class** operator introduces the following bindings:

(1) The identifiers Department and Employee are bound to new types isomorphic to tuples.
(2) The class identifiers Departments and Employees are bound to modifiable sequences of values of type Department and Employee, respectively.
(3) The identifiers mkDepartment and mkEmployee are bound to two primitive functions, automatically declared, which differ from similar functions for abstract types in that every time they are applied, new objects are created and automatically inserted in front of the associated sequences if the specified constraints are not violated. The constructed elements are also the values returned by these functions.

The above declaration defines the structure of the objects together with a few constraints, some of which are predefined constraints on sequences, to be tested when an instance is created or modified:

(1) The **key** constraint asserts that elements of a class must differ in the value of certain constant attributes. Note that if the **key** constraint is not specified, the insertion will be made even though the values of the attributes are equal to those of another object already present in the class. That is, elements of classes are always distinct objects, but the construction of an element will fail when the constraints are violated.

Other constraints are specified directly in the definition of the element type:

(2) Only attributes with a **var** type can be modified.
(3) Only modifiable attributes with an **optional** type can be left unspecified when an element is created.
(4) A derived attribute such as NameOfDept is used to model a mapping from the employees to the department where they are employed, while the property Employees in Departments is used to model a *part-of* relationship, which implies the following dependency constraint: an employee cannot be eliminated from the database as long as he or she belongs to a department.

Since the name of a class denotes the sequence of all the current elements present in the database, all the operators on sequences can be applied to classes. In addition to these operators, the following is also provided:

get ClassId **with** Condition

This is another operator on sequences: it returns the only element in a sequence which satisfies the condition. Otherwise, a failure is generated.

6.2 Subclasses

Subclasses and type hierarchies are the features provided by Galileo to support the abstraction mechanism of IS-A hierarchies, originally proposed in the context of semantic networks, and considered nowadays as an essential requirement for a language supporting semantic data model features [35].

There are, however, differences between IS-A hierarchies and the type hierarchies introduced in the previous section:

(1) The subtype notion in Galileo refers to a static aspect of the language, and has been introduced to establish a compatibility rule among all the possible values of a type and those of its supertypes.
(2) An IS-A hierarchy (e.g., Students IS-A Persons) involves two different notions. First, it establishes an existence constraint among the elements of Students and Persons present in the database: the elements of Students are always a subset of the elements of Persons (extensional notion). Second, it establishes a subtype hierarchy between the type of the elements of Students and Persons. Therefore, an element of Students can be used as an argument of any operation defined for elements of Persons (intensional notion).

In Galileo, the two notions behind the IS-A hierarchy are expressed with two distinct mechanisms: the type hierarchy, to deal with the intensional aspect, and the subclass, to deal with the extensional aspect. This distinction increases the modeling capability of the language because it allows the use of the type hierarchy independently of the subclass mechanism.

There are three ways of defining subclasses: by *subset*, *partition*, or *restriction*.

A *subset* class with elements of type T contains those elements of the parent class that have been included explicitly in the subclass with the proper operator in T.

A *partition* class is like a subset class, but it enforces the additional constraint that its elements are not included in another subclass of the same partition.

A *restriction* class contains all the elements of the parent class that satisfy some predicate, which is evaluated at the time of element construction. This predicate cannot be defined over modifiable or derived values.

In all cases, when a new element is added to a subclass it then also becomes an element of the parent class. In the case of restriction classes, a new element must also satisfy the restriction predicate.

Finally, the operator

remove Expression1, . . . , ExpressionN

is provided to eliminate objects from a class and from its subclasses, and return the value nil only if the objects are not used as components of other elements. Otherwise, a failure is generated. "Expression i" must evaluate to a sequence of elements.

The type of the elements of a subclass must be a subtype of the element type of the parent class. New attributes can be added with the **and** operator or old attributes can be redefined with the **ext** operator, but the following restrictions must be satisfied:

(1) Nonoptional attributes may be added only when a subclass is defined as a subset or partition.
(2) When a subclass is defined by restriction, then only derived, optional, or default attributes can be added.

Subclasses can also be defined from more than one parent class, with the restriction that the type of the elements must be a subtype of the element type

of each parent class. An element of a subclass is always an element of all its parent classes. Some examples follow to clarify these points:

```
PublicEmployees restriction of Employees class
  PublicEmployee ↔ is Employee
```

The elements are the same as the Employee class.

```
DowntownDepartments restriction of Departments
  with Address = "Downtown"
class
  DowntownDept ↔
    (is Department
     ext ManagerSalary := derived Salary of (at Manager))
```

The elements of the DowntownDepartments class are all the departments in Downtown.

```
Managers partition of Employees
    with Secretaries, Craftsmen class
  Manager ↔ (is Employee and Bonus: num)


Secretaries partition of Employees
    with Managers, Craftsmen class
  Secretary ↔ is Employee


Craftsmen partition of Employees
    with Secretaries, Managers class
  Craftsman ↔ is Employee


Carpenters subset of Craftsmen class
  Carpenter ↔ is Craftsman


Bricklayers subset of Craftsmen class
  Bricklayer ↔ is Craftsman
```

The Employees are partitioned into three disjoint subsets, while the Craftsmen have been refined into two overlapping subsets of instances. In all the above cases, the classes must be populated explicitly.

The predicate **alsoin** is provided to check whether or not an object of one class also belongs to a subclass:

```
Expression alsoin Subclass
```

Expression must evaluate to an object of a class.

The following operator is used to include an element of a class in a subclass with elements of type *T*:

```
inT (Expression1, Expression2)
```

Expression1 must evaluate to the object to be included in the subclass, while Expression2 must evaluate to a value of the representation type *T*. The operator

checks that the values of the corresponding attributes of Expression1 and Expression2 are the same. Expression2 can be omitted when an object of a subclass has the same attributes as an object of the superclass.

Finally, to operate on an object of a parent class as if it were the element of a subclass, the object must be retyped with the following operator:

Expression **likein** Subclass

Expression must evaluate to an object of a class. The result is the object as member of Subclass. This operator is needed due to the static type checking discipline.

7. ENVIRONMENTS AS A MODULARIZATION MECHANISM

The languages hitherto proposed for conceptual modeling do not provide features to help the designer to develop and test a schema incrementally or to express the overall structure of a schema in terms of smaller related parts. This issue has been addressed in Galileo by using the environment, whch is a denotable value, as a modularization mechanism [5]. As will be shown in the sequel to this paper, the environment operators previously defined can be used to structure a schema in a way similar to that suggested for theories by Burstall and Goguen in their specification language Clear [23].

Another use of environments is to deal with data and operations as a single unit which can be accessed by programs. This problem has also been addressed in ADAPLEX with a specialized form of Ada packages [49]. In fact, a drawback to commercial DBMSs is that no kind of procedural knowledge can be described in the schema, whether "derived" information or application domain oriented operations. In other words, in these systems data can be shared, but the procedural knowledge cannot: it must be embedded in the applications. The inclusion of the operations in the schema has the following advantages:

(1) The same operations on the database are not duplicated in all the programs that need them.
(2) The database schema does reflect all the knowledge available about the application domain. In particular, the schema contains not only the description of the structure of the objects and the constraints, but also the operations on the objects, which complete their semantics.
(3) It is possible to constrain user programs to operate on the database through a set of predefined operations, especially designed to include critical design choices, such as integrity preservation.

Environments also have other useful applications. First, it is the mechanism used by Galileo to deal with persistence without resorting to specific data types, such as files of programming languages. Second, to deal with evolving applications, the environment is used to establish explicitly the way in which new applications interact when they use common data. Finally, the environment is used to define application-oriented views of data in a similar way to the view mechanisms of DBMSs.

7.1 Persistence

Temporary values exist in the system only during the execution of the expression in which they are defined. None of the abstraction mechanisms described previously have the property of defining persistent values. For instance, user programs may also contain class definitions, if temporary classes must be kept while running an application. To deal with persistence, a global environment is assumed in which all values are automatically maintained. Such an environment is managed by the system that supports the language. For other approaches to the treatment of persistence as an orthogonal property of data, see [10].

The global environment is extended by adding new bindings with the command **use**. In fact, for user protection, a warning is generated if **use** is used with identifiers already bound in the current environment. Instead of having a single set of unrelated definitions and values, as imposed by the interactive approaches of LISP top level and APL workspace, the user can fruitfully employ the environment mechanism to structure the global environment. For instance, the following is the definition, at top level, of an environment Personnel with two permanent classes (for brevity, defined operations are omitted):

```
use
Personnel :=
  (rec Departments class
    Department ↔
      (Name: string
      and Manager: var Employee
      and Budget: num)
      key (Name)
  and Employees class
    Employee ↔
      (Name: string
      and Salary: num
      and Dept: var Department)
      key (Name));
```

Each expression is evaluated inside an environment, initially the global one, called the *current environment.* Any environment that can be accessed from the global environment can become the current one with the command "**enter** Environment", while to return to the global environment there is the command **quit**. Since the language is expression-based, it is possible in the current environment to evaluate any expression by simply typing it. For example, assuming that the classes in Personnel have already been populated, a simple interactive session is:[1]

```
enter Personnel:
```

To get the names of all the employees with a salary less than the average salary

[1] A more elaborate session is reported in [4].

of their department:

```
for x in Employees
  with Salary of x
    < avg(for y in Employees
          with at Dept of x = at Dept of y
          do Salary of y)
do Name of x;
```

To add a new employee to the Research department:

```
mkEmployee
  (Name := "Brown"
  and Salary := 4
  and Dept := get Departments with Name = "Research");
```

7.2 Encapsulation

Another use of the environment mechanism is to model a schema as a set of interrelated units. Each unit encapsulates data and operations that are closely related. For instance, let us assume that we are interested in describing as distinct units data relevant to the planning and administration departments of our hypothetical firm, although these departments share data of the environment Personnel:

```
use Planning :=
  (Personnel
  and Projects class
    Project ↔
      (Name: string
       and Budget: num)
      key (Name));
use Administration :=
  (Personnel
  and Suppliers class
    Supplier ↔
      (Name: string
       and Address: var string
       and Credit: var num)
      key (Name));
```

Note that, because of the semantics of environment operators, the Personnel environment is shared by Planning and Administration, so that any updating of a class in any environment will be reflected in all the others.

7.3 Refinements

It is possible to start with one environment and to generate others by extending the environment with new definitions. Thus, data concerning the same application are visible at different levels of detail.

```
use DetailedPersonnel :=
  (Personnel
  and Branches class
    Branch ↔
      (Name: string
       and Address: string
       and Other: string)
      key (Name)
```

```
ext Special Employees subset of Employees class
  SpecialEmployee ↔
    (is Employee
    and PrivateData: string));
```

7.4 View Modeling

To provide controlled access to the database, it is possible to give a different view of an environment by excluding some of its data or operations.

```
use OnlyDepartments := Personnel drop Employees
```

In OnlyDepartments, Employees are not visible, while in the following environment only the names of the employees and the names of the departments where they work can be accessed:

```
use EmployeesView :=
  (use Personnel
  in Employees :=
    derived for e in Employees
      do (e ext NameOfDept := Name of Dept)
      drop Dept, Salary);
```

The expression "Id := **derived** Expression" denotes an environment in which the only association is between the Id and a virtual value, which is obtained by evaluating Expression every time the value of Id is requested. All the operators used to query a class can be applied to Employees, which therefore behaves like a view of relational database.

7.5 Logical Independence

The environment operators allow the designer to make applications independent from changes in an environment, as long as the old view of the database is derivable from the redefined environment. For instance, let us assume that an application program was designed to work in the DetailedPersonnel environment on Branches of a certain area, "Downtown", to retrieve data. The database was then extended to include Branches in other areas, with the elements type redefined as

```
Branch ↔
  (Name: string
   and Address: string
   and Area: string
   and Other: string)
```

In order to make the old program independent of these changes, it can be used in the following environment:

```
use NewDetailedPersonnel :=
  (DetailedPersonnel
   ext Branches :=
    derived for b in Branches
      with Area = "Downtown"
      do b drop Area);
```

8. TRANSACTIONS AND FAILURE HANDLING

Every top level Galileo expression is a *transaction.* That is to say, it is considered an atomic action against the database: once invoked, it either completes all its operations or behaves as if it were never invoked. Transactions may fail due either to a hardware or software failure or to a run-time program error. In Galileo it is possible to cause such an event, and also to sense its occurrence so as to perform an appropriate action. The failure of a transaction causes an interruption of the normal control flow, and, in addition, all updatings from the beginning of the transaction are undone.

A transaction can be either *simple* or *compound.* Each expression typed in at top level by the user is a simple transaction. Therefore, if the expression fails, the persistent data are unaffected. However, if more than one top level expression must be considered as a single transaction, the expressions must be enclosed in "transaction brackets": **transaction** and **end_transaction**. A compound transaction is a sequence of top level expressions enclosed in such brackets.

Since any operation whether predefined or defined that is accessible to the user may be applied as a simple transaction, whenever the schema designer defines operations, he or she is in fact defining transactions. As a consequence, transactions can be nested by defining a new function as a composition of predefined ones: an action, atomic at a higher level of abstraction, may be decomposed into subatomic actions to perform, for example, a stepwise updating of the database [30]. A failure of inner transactions can be controlled, and alternative transactions can be started to achieve the desired effect. Consider, for example, the case of booking a tour with an airline reservations system. Even if the reservation of single parts of the tour succeeds, unless all the tour has been reserved, the effects of previous operations must be revoked, and a new attempt could be made with a different airline, or with a different schedule. The different attempts should be treated as alternative transactions, and the outermost one should fail only if all attempts fail. Another important advantage of nested transactions is the ability to define transactions not knowing the context in which they might be used [9].

The linguistic construct for handling failures has a block structure, unlike the usual proposed **commit** and **abort** statements [30]: "Expression **if_fails** Expression". If the first expression fails, its effects are undone, and the value of the whole construct is that of the second expression. Otherwise, it is that of the first one with the effects preserved.

Failures have associated with them a string that can be used for a selective handling of failures with the **case_fails** construct. For failures which occur during the execution of primitive operations, the string returned is the name of the operation. The user can generate a failure with the expression "**failwith** string" or with **fail**, which is equivalent to "**failwith** "fail"". When a failure occurs, the normal execution path is interrupted, control is passed to the first surrounding failure handler, and the effects are undone. If no handler is present, the top level expression fails, all its effects are undone, an error message is printed, and the execution terminates. Let us consider an example with the

selective failure handler:

```
Employee class
  Employee ↔
    (Name: string
     and Salary: num
     and Dept: Department)
    key (Name)
    assert with "LowPay" Salary < Minimum
    assert with "HighPay" Salary < (Budget of Dept)/10
ext rec NewEmployee(AName: string, ASalary: num, ADept: string):Employee :=
  mkEmployee
    (Name := AName
     and Salary := ASalary
     and Dept := get Departments with Name = ADept)
  case_fails
    ["LowPay"]
        NewEmployee(AName, Minimum, ADept) |
    ["HighPay"]
      NewEmployee(AName,
        (Budget of get Departments with Name = ADept)/10,
        ADept)
```

9. CONCLUSIONS

A strongly-typed programming language for database applications has been presented. Unlike other proposals, which integrate a relational data model into a conventional, general-purpose programming language, e.g., Pascal, [8, 40, 43, 46, 52], we have integrated into the framework of the programming language Edinburgh ML [29], a strongly-typed interactive language, features to support semantic data model abstraction mechanisms (classification, aggregation, and specialization) as well as abstraction mechanisms of modern programming languages (types, abstract types, and modularization).

The approach adpoted is therefore closer to that of ADAPLEX, which extends Ada with new features to support databases modeling [49]; although the features included in Galileo, notably the type hierarchies, are not ad hoc for databases, but can be used independently. This approach was preferred for two reasons.

First, we were interested in studying a uniform approach towards the design of a modern strongly-typed programming language, which would include features to support semantic data models. We believe that this paper provides evidence of how types, abstract types, type hierarchies, classification, aggregation, specialization, and modularization can be integrated in an expression-based language that is statically type-checkable. In particular, we have shown the effectiveness of the environment, a novel abstraction mechanism, in the context of conceptual modeling, for structuring complex applications and for view modeling.

Second, we were interested in developing an interactive database designer's workbench, which would integrate a set of tools for creating, testing, and implementing on a traditional DBMS a database design [2]. Since, in the short term, we have mainly been interested in using this aid for conceptual modeling, we have found it more convenient to design a new language for dealing with the specific problems in this area. We have already implemented a prototype version

of the system, called Dialogo, which presently supports a significant subset of Galileo [4]. Tools are available to edit a conceptual schema, query the definitions, and load and query test data. An interesting feature of Dialogo is that it is based on a top level cycle in which a Galileo expression from the user is accepted, executed, and the result displayed while the effect of the user expression on the database is permanently preserved. An expression may be the invocation of a single predefined function or any complex expression of the language.

Future studies on Galileo will proceed along the following lines:

(1) *Extensions.* We will extend the language to provide (a) a form-oriented, input/output interface; (b) a process construct to model interactions with the users and database evolution, with an approach similar to that adopted in TAXIS.
(2) *Implementation.* The Dialogo system is being reimplemented by extending the present implementation of the ML compiler, available on a VAX 11/780 running the UNIX[2] operating system.
(3) *Applications.* With the new implementation of Dialogo, it will be possible to effectively experiment with the design of database applications using Galileo. This will also provide the opportunity to test the tools available in our designer's workbench against the demands of specific user environments.

ACKNOWLEDGMENTS

We are indebted to M. E. Occhiuto, who contributed to the design of a preliminary version of Galileo, and to the members of the Galileo Project, M. Capaccioli, F. Giannotti, B. Magnani, D. Pedreschi, and M. L. Sabatini, for their constructive criticisms. Also, many thanks to A. Borgida, S. Gibbs, D. Lee, A. Mendelzon, J. Mylopoulos, B. Nixon, and I. Reichstein for their helpful suggestions in improving a previous version of this paper, at the time A. Albano was visiting professor at the Computer Science Department of the University of Toronto. The paper has also benefitted from the constructive comments made by the referees.

REFERENCES

1. ABRIAL, J.R. Data semantics. In *Data Management Systems*, J. K. Klimbie and K. L. Koffeman, Eds., North-Holland, Amsterdam, 1974, 1–60.
2. ALBANO, A., AND ORSINI, R. An interactive integrated system to design and use data bases. In *Proceedings Workshop on Data Abstraction, Data Bases and Conceptual Modelling, ACM SIGMOD Special Issue 11*, 2 (1981), 91–93.
3. ALBANO, A., OCCHIUTO, M.E., AND ORSINI, R. A uniform management of persistent and complex data in programming languages. In *Infotech State of Art Report on Databases*, M.P. Atkinson, Ed., Series 9, No. 4, Pergamon Infotech, 1981, 321–344.
4. ALBANO, A., AND ORSINI, R. Dialogo: An interactive environment for conceptual design in Galileo. In *Methodology and Tools for Database Design*, S. Ceri, Ed., North-Holland, Amsterdam, 1983, 229–253.
5. ALBANO, A. Type hierarchies and semantic data models. *ACM Sigplan '83: Symposium on Programming Language Issues in Software Systems* (San Francisco, 1983), 178–186.
6. ALBANO A., CAPACCIOLI, M., AND ORSINI, R. La definizione del Galileo (Versione 83/6). Rapporto Tecnico DATAID N.20, Pisa, 1983.

[2] UNIX is a trademark of Bell Laboratories.

7. ALBANO, A., CAPACCIOLI, M., OCCHIUTO, M.E., AND ORSINI, R. A modularization mechanism for conceptual modeling. In *Proceedings 9th International Conference on VLDB* (Florence, Italy, 1983), 232–240.
8. AMBLE, T., BRATBERGSENGEN, K., AND RISNES, O. ASTRAL, a structured and unified approach to database design and manipulation. In *Data Base Architecture*, G. Bracchi and G.M. Nijssen, Eds., North-Holland, Amsterdam, 1979, 240–257.
9. ATKINSON, M.P., CHISHOLM, K.J., AND COCKSHOTT, W.P. The new Edinburgh persistent algorithmic language. In *Infotech State of Art Report on Databases*, M.P. Atkinson, Ed., Series 9, No. 4, Pergamon Infotech, 1981, 299–318.
10. ATKINSON, M.P., BAILEY, P.J., CHISHOLM, K.J., COCKSHOTT, W.P., AND MORRISON, R. An approach to persistent programming. *Comput. J. 26*, 4 (1983), 360–365.
11. BALTZER, R. An implementation methodology for semantic database models. In *Entity Relationship Approach to System Analysis and Design*, P.P. Chen, Ed., North-Holland, Amsterdam, 1980, 433–444.
12. BARRON, J. Dialogue organization and structure for interactive information systems. M.Sc. thesis, Dept. of Computer Science, Univ. of Toronto, 1980.
13. BILLER, H. AND NEUHOLD, E.J. Semantics of databases: The semantics of data models. *Inf. Syst. 3* (1978), 1–30.
14. BORGIDA, A.T., MYLOPOULOS, J., AND WONG, H.K.T. Methodological and computer aids for interactive information systems design. *Automated Tools for Information System Design*, H.J. Schneider and A. Wasserman, Eds., North-Holland, Amsterdam, 1982.
15. BORGIDA, A. Features of languages for the development of information systems at the conceptual level. *IEEE Softw.* (1984), to appear.
16. BREUTMAN, B., FALKENBERG, E., AND MAUER, R. CSL: A language for defining conceptual schemas. In *Data Base Architecture*, G. Bracchi and G.M. Nijssen, Eds., North-Holland, Amsterdam, 1979, 237–256.
17. BRODIE, M.L. The application of data types to database semantic integrity. *Inf. Syst. 5*, 4 (1980), 287–296.
18. BRODIE, M.L., AND ZILLES, S.N. Eds. *Proceedings Workshop on Data Abstraction, Data Bases, and Conceptual Modelling, ACM SIGMOD Special Issue 11*, 2 (1981).
19. BRODIE, M.L. On modeling behavioral semantics of databases. In *Proceedings 7th International Conference on VLDB* (Cannes, 1981), 32–42.
20. BRODIE, M.L., MYLOPOULOS, J., AND SCHMIDT, J.W. Eds. *On Conceptual Modeling: Perspectives from Artificial Intelligence, Databases, and Programming Languages*, Springer Verlag, New York, 1984.
21. BUBENKO, J.A. Information modeling in the context of system development. In *IFIP Congress 1980*, North-Holland, Amsterdam, 1980, 395–411.
22. BUNEMAN, P., AND FRANKEL, R.E. FQL—a functional query language. In *Proceedings of ACM SIGMOD Conference* (Boston, Mass., 1979), 52–58.
23. BURSTALL, R.M., AND GOGUEN, J.A. Putting theories together to make specifications. In *Proceedings IJCAI* (Boston, Mass., 1977), 1045–1058.
24. CAPACCIOLI, M. La Semantica Denotazionale del Galileo. Tesi di laurea in Scienze dell'Informazione, Univ. di Pisa, Italy, 1983.
25. CARDELLI, L. A semantics of multiple inheritance. In *Semantics of Data Types*, G. Kahn, D.B. MacQueen, and G. Plotkin, Eds., Lecture Notes in Computer Science, Vol. 173, Springer Verlag, New York, 1984, 51–67.
26. CERI, S., PELAGATTI, G. AND BRACCHI, G. Structured methodology for defining static and dynamic aspects of data base applications. *Inf. Syst. 6*, 1 (1981), 31–45.
27. CERI, S., Ed. *Methodology and Tools for Database Design*, North-Holland, Amsterdam, 1983.
28. GORDON, M. *The Denotational Description of Programming Languages. An Introduction*, Springer Verlag, New York, 1979.
29. GORDON, M., MILNER, R., AND WADSWORTH, C. *Edinburgh LCF*, Lecture Notes in Computer Science, Vol. 78, Springer Verlag, New York, 1979.
30. GRAY, J. The transaction concept: Virtues and limitations. In *Proceedings 7th International Conference on VLDB* (Cannes, 1981), 144–154.
31. HAMMER, M., AND BERKOWITZ, B. DIAL: A programming language for data intensive applications. In *Proceedings of ACM SIGMOD Conference*, (Santa Monica, Calif., 1980), 75–92.

32. HAMMER, M., AND MCLEOD, D. Database description with SDM: A semantic database model. *ACM Trans. Database Syst. 6*, 3 (1981), 351–386.
33. KENT, W. Limitations of record-based information models. *ACM Trans. Database Syst. 4*, 1 (1979), 107–131.
34. LUM, V., ET AL. 1978 New Orleans Data Base Design Workshop Report. In *Proceedings 5th International Conference on VLDB* (Rio de Janeiro, 1979), 328–339.
35. MCLEOD, D., AND KING, R. Semantic database models. In *Principle of Database Design*, S.B. Yao, Ed., Prentice-Hall, 1984.
36. MILNER, R. A theory of type polymorphism in programming. *J. Comput. Syst. Sci. 17*, (1978), 348–375.
37. MYLOPOULOS, J., BERNSTEIN, P.A., AND WONG, H.K.T. A language facility for designing database-intensive applications. *ACM Trans. Database Syst. 5*, 2 (1980), 185–207.
38. NAVATHE, B.S. Information modeling tools for data base design. Panel on Logical Database Design (Fort Lauderdale, Fla., 1980).
39. ROUSSOPOULOS, N. CSDL: A conceptual schema definition language for the design of data base applications. *IEEE Trans. Softw. Eng. SE-5*, 5 (1979), 481–496.
40. ROWE, L.A., AND SHOENS, K.A. Data abstraction, views and updates in RIGEL. In *Proceedings ACM SIGMOD Conference* (Boston, Mass., 1979), 71–81.
41. SABATINI, L. La Semantica Statica del Galileo. Tesi di laurea in Scienze dell'Informazione, Univ. di Pisa, Italy, 1982.
42. SCHMIDT, J.W. Type concepts for database definition. In *Database: Improving Usability and Responsiveness*, B. Schneidermann, Ed., Academic Press, New York, 1978, 215–244.
43. SCHMIDT, J.W., AND MALL, M. *Pascal/R Report.* Univ. of Hamburg, Fachbereich Informatik, Rep. 66, Jan. 1980.
44. SHAW, M. The impact of abstraction concerns on modern programming languages. *Proc. IEEE 68*, 9 (1980), 1119–1130.
45. SHIPMAN, D.W. The functional data model and the data language DAPLEX. *ACM Trans. Database Syst. 6*, 1 (1980), 140–173.
46. SHOPIRO, J.E. A programming language for relational databases. *ACM Trans. Database Syst. 4*, 4 (1979), 493–517.
47. SMITH, J.M., AND SMITH, D.C.P. Database abstraction: Aggregation and generalization. *ACM Trans. Database Syst. 2*, 2 (1979), 105–133.
48. SMITH, J.M., AND SMITH, D.C.P. A database approach to software specifications. In *Software Development Tools*, W.E. Riddle and R.E. Fairley, Eds., Springer Verlag, Berlin, 1979, 176–200.
49. SMITH, J.M., FOX, S., AND LANCERS, T. Reference manual for ADAPLEX. Tech. Rep. CCA-81-02, Computer Corporation of America, Jan. 1981.
50. TEICHROEW, D., AND HERSHEY, E.A. PSL/PSA: A computer-aided technique for structured documentation and analysis of information processing systems. *IEEE Trans. Softw. Eng. SE-3*, 1 (1977), 41–49.
51. TENNENT, R.D. *Principles of Programming Languages.* Prentice-Hall International, London, 1981.
52. WASSERMAN, A.I. The data management facilities of PLAIN. In *Proceedings of the ACM SIGMOD Conference* (Boston, Mass., 1979), 60–70.
53. WEBER, H. A software engineering view of data base systems. In *Proceedings 4th International Conference on VLDB* (Berlin, 1978), 36–51.
54. YAO, S.B., NAVATHE, S.B., AND WELDON, J.L. An integrated approach to logical database design. In *Proceedings NYU Symposium on Data Base Design*, (1978), 1–14.

Received April 1984; revised January 1985; accepted February 1985