A MODULARIZATION MECHANISM FOR CONCEPTUAL MODELING

A. Albano (*), M. Capaccioli (+), M.E. Occhiuto (+) and R. Orsini (*)

(*) Dipartimento di Informatica, C.so Italia, 40, 56100 Pisa, Italy
(+) Systems & Management S.p.A., V.lo S. Pierino 4, 56100 Pisa, Italy

Abstract

The abstraction mechanisms of Semantic Data Models - aggregation, classification and generalization - are considered the basic features to be supported by conceptual languages, i.e., programming languages with high-level constructs for database applications. This paper shows that conceptual languages should also provide a modularization mechanism as another feature to achieve a more adequate database modeling capability. Such a mechanism is required to organize structural and procedural aspects of a complex schema in smaller, interrelates units. A proposal is presented in the framework of Galileo, a strongly-typed, interactive, conceptual language designed specifically for database applications.

1. INTRODUCTION

Recently, database research workers have been paying attention to the design of conceptual languages, i.e., programming languages for database applications that support the abstraction mechanisms of Semantic Data Models, besides the traditional abstraction mechanisms for temporary data.

A Semantic Data Model is a set of abstraction mechanisms to describe the structure of databases: the data abstractions, together with the implicit constraints and the associated operations, are explicitly intended to represent naturally and directly certain types of real-world information. A survey and analysis of the motivations for this new generation of data models is reported in (McLeod 82). Well known examples of conceptual languages are TAXIS (Mylopoulos 80), DIAL (Hammer 80), and ADAPLEX (Smith 81).

Still, an important open problem in conceptual language design is which features should be integrated into a programming language to achieve adequate database modeling capabilities. For instance, different opinions exist on the use of data types and on which features a programming language should have to support the basic abstraction mechanisms of Semantic Data Models, i.e., aggregation, classification and generalization (Albano 83c, Brodie 80, 81). Examples of these different trends are: TAXIS, a programming language based on a procedural semantic network formalism; DIAL, which has evolved from SDM (Hammer 81), and ADAPLEX, based on programming languages with data types extended by a class construct; RM/T (Codd 79) and SHM (Smith 79), proposed to extend the relational model outside the framework of a specific programming language.

There is, however, another issue that deserves more attention (Wong 77, Mylopoulos 81): which features are needed to organize structural and procedural aspects of a complex schema in smaller, conceptually meaningful, interrelated units. This requirement is motivated by the following considerations:

This work was supported in part by the Consiglio Nazionale delle Ricerche, Progetto Finalizzato Informatica, Obiettivo DATAID, and in part by Ministero della Pubblica Istruzione.

a. A schema should include the definition of structural aspects of a database together with the procedural aspects. That is to say, a database should appear to the user as a module allowing access to both data and operations on the database.

b. One aspect of the schema complexity in the large number of details. An effective way of mastering this difficulty is "Taxonomic Specification", suggested in (Borgida 82). It is a stepwise methodology, based on the generalization abstraction mechanism, which suggests organizing the definitions of a complex schema into hierarchies by a controlled introduction of details. To support this methodology, it is useful to have a construct in the language to organize the schema in distinct, but related, units corresponding to the different levels of refinements. In this way the units can also be used to give zoomed versions of the application model.

c. Another reason to have a schema organized in units relies on the need of modeling complex applications involving different enterprise sectors. Instead of having a single schema which does not favor the recognition of the relationships among its parts, a database description organized in terms of interrelated units makes its structure explicit, and therefore the description results in a more natural model of the application. In general, different units do not model independent aspects of the application, but they can share common data and operations. This notion of a schema organized in parts is more related to the problem of mastering complex software by modular decomposition, than to the view modeling in DBMSs.

d. A view modeling capability is another requirement for conceptual languages. A view mechanism should be widely variant in capability: it might only allow access to a subset of the schema definitions (data and operations), or it might allow the specification of more complex mappings among the objects in the view and those in the schema. In both the cases, the user of a view would operate on the accessible objects as if they were those of a schema, except for the operations explicitly excluded. A view mechanism should not be confused with the capability discussed at the previous point; such a mechanism is complementary to those available for schema definition.

Proposals exist in the literature addressing some of the previous issues, but none of them casts the solution in the context of a specific conceptual language.

The possibility of organizing complex schema in parts has been considered for relational databases in ASTRAL (Amble 79). ASTRAL's module mechanism is used to organize a relational schema hierarchically, so that if a module A is below a module B, everything exported from B can be imported in A. Other proposals have been given in the area of Artificial Intelligence, but for different purposes. The problem of interest here is the modeling of hypothetical worlds and belief spaces: "context", or "spaces", are organized hierarchically in PLANNER-like languages, as in ASTRAL, but data are not shared because contexts evolve indipendently (Montangero 78). An interesting use of these mechanisms appears in (Abrial 74, Hendrix 75). In Abrial's proposal, a global context is provided where data are stored permanently. New contexts can then be created with the possibility of specifying whether they are permanent or temporary. Database updates in a context can be reflected in higher contexts specified by the user. In (Hendrix 75) spaces have been proposed as an extension of contexts, in that they can be structured into an acyclic graph rather then a tree.

View mechanisms are usually present in DBMSs. Relational systems are more powerful than those based on the DBTG proposal because a view is more general than a subschema, in that any relation derivable by an expression can be queried as if it were a relation of a view. This possibility is also present in some relational database programming languages, in which update operations may be included in a module definition (Rowe 79, Shopiro 79, Wasserman 79). Among conceptual languages, a similar approach is adopted by ADAPLEX, with modules modeling view, while TAXIS and DIAL provide, respectively, the "script" and "port" mechanisms to model interaction with the user. These mechanisms restrict the objects accessible, but are not used in modeling views.

The purpose of this paper is to propose a structuring mechanism for conceptual modeling. The presentation will be centered around Galileo, a strongly typed, interactive, programming language (Albano 83a). In the next section a brief overview of the language is given. Section 3 describes the notion of environment in Galileo and Section 4 contains examples showing how databases can be described in a structured way.

2. OVERVIEW OF Galileo

A complete description of the language is beyond the scope of this paper, and may be found in (Albano 83a), and, together with the denotational semantics, in (Capaccioli 83). A preliminary implementation of a subset of Galileo has been described in (Albano 83b). Presently, a more efficient implementation is in progress on a VAX 11/780 running the UNIX (*) operating system.

Galileo is a programming language supporting the abstraction mechanisms of Semantic Data Models and the data abstractions of programming languages.

The main features of Galileo are:

a. The language is expression oriented. Each construct is applied to values and returns a value.

b. Every denotable value of the language possesses a type, which defines a set of values sharing common characteristics, together with the operators which can be applied to these values. Besides the usual predefined types of programming languages, the type constructors available are: tuple (record), sequence, discriminated union, function, modifiable value (reference), and abstract types.

c. Galileo's type system supports the notion of a "type hierarchy" (Albano 83c). If a type T is a subtype of a type T', then a value of type T can be used as argument of any operation defined on values of type T', but not vice versa. The subtype relation is a partial order.

d. Every Galileo expression has a type. In general, the type of any expression can be statically determined. Every type violation can be detected by textual inspection (static type checking). Type information is only used during the static analysis of expressions, and is ignored at run-time, when testing is required for constraints only. This is made possible since Galileo has a secure type system: 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 expected type.

e. Galileo provides a mechanism, called "classes", to represent real world entities in a Galileo database by classification and aggregation. 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 to add or delete elements from a class are implicitly defined. Classes can be defined by subsetting, partitioning, and restricting other classes. They are used to model alternative ways of looking at the same entities, including the IS_A hierarchy.

3. ENVIRONMENTS

To understand the modularization feature we will illustrate, it is useful to first understand what are Galileo's environments.

For any expression, the meaning of identifiers in use is given with respect to the current environment. An environment is composed of two parts: a "type component" and a "value component".

The type component of an environment is a set of associations (identifier, type), and it is used to establish which type a type identifier denotes. The value component of an environment is a set of associations (identifier, value) and it is used to establish which value an identifier denotes.

Environments are defined using the following operators, which work on both the components. Only those properties of the operators necessary to understand how modularization is supported in Galileo are presented. Examples will be presented in the next section.

( )

denotes the empty environment, in which there are no associations.

Identifier := Expression

denotes the environment in which the only association is between Identifier and the value denoted by Expression, which can be of any kind, including functions or environments.

Identifier := **derived** Expression

(*) UNIX is a Trademark of Bell Laboratories.

denotes the environment in which the only association is between Identifier and a virtual value, which is obtained by evaluating Expression every time the value of Identifier is requested. Since Galileo has a discipline of static scope binding, Expression is always evaluated in the definition environment.

In the previous cases the created environment had an empty type component. In fact, no definition of types appeared there.

type TypeIdentifier := TypeExpression

denotes the environment in which the only association is between TypeIdentifier and the type denoted by TypeExpression. With this form a concrete type, i.e., non abstract, is defined; TypeIdentifier is only an abbreviation for the structure it represents.

```
type TypeIdentifier <=>
      TypeExpression
      assert BooleanExpression
```

denotes an environment where:

a. TypeIdentifier is bound to a new abstract type, with a domain isomorphic to the domain of the representation type, given by TypeExpression, possibly restricted by the assertion.

b. The identifier mkTypeIdentifier and repTypeIdentifier are bound to two primitive functions that map values, respectively, of the representation type into the abstract type and vice versa. In the assert clause, BooleanExpression is a condition on values of the representation type. The assertion is a dynamic constraint and is controlled by the constructor mkTypeIdentifier. When an assertion is violated the operation that caused it fails.

```
type TypeIdentifier <->
      TypeExpression
      assert BooleanExpression
```

If Opi, ..., Opn are predefined operators on values of the concrete type TypeExpression, this expression denotes an environment containing all the bindings constructed by <=> and, in addition, Opi, ..., Opn redefined to operate on values of TypeIdentifier.

Identifier **class** AbstractType

denotes an environment constituted by the bindings exported by AbstracType, and, in addition, the binding (Identifier, empty sequence of elements of the abstract type). The constructor of the abstract type values has the additional property that a constructed value also becomes an element of the class.

We now show the operators used to define environments in terms of others. In the following, A and B stand for expressions denoting environments.

A **and** B

denotes an environment with all the bindings of A and of B. Both A and B are evaluated independently in the current environment and must not have common identifiers.

A **ext** B

denotes an environment with all the bindings of B and those of A not redefined in B. A is evaluated in the current environment, while B is evaluated in the current environment extended with the bindings of A.

rec A

denotes an environment with bindings of A, evaluated in the current environment extended by the bindings of A. This operator is required for mutually dependent definitions.

A **drop** Identifier

denotes an environment containing all the bindings of A except the one with binder Identifier.

A **rename** Identifier **in** NewIdentifier

denotes an environment with the bindings of A but the binder Identifier is renamed as NewIdentifier.

An environment A can be used in the evaluation of an expression, extending the current environment with the bindings of A:

use A **in** Expression

4. STRUCTURING DATABASES WITH ENVIRONMENTS

Using the analogy of software design, the structuring mechanisms for conceptual languages hitherto proposed can be called mechanisms for "designing databases in the small". They help to structure a schema as a single unit, but do not help to solve the problem of "designing in the large", where complex applications are involved. In the software engineering area this need has been considered, and nowdays programming languages provide features to decompose large software projects into smaller units.

This issue has been addressed in Galileo using the environment to structure a schema into a set of smaller, related parts, sharing common data and operations. As shown in the previous section, environments can be manipulated by a set of operators. To structure databases and to model views, the useful operators are: "and" and "ext" to extend or combine environents; "drop" to exclude definitions; "rename" to rename identifiers. In the sequel it will be shown by examples the effectiveness of the approach, which has the following advantages. Firts, the environment may be used to deal with data and operations as a single unit, accessible to users. Secondly, it may be used to deal with data persistence without resorting to specific data types, such as the files of programming languages. Thirdly, the environment may be used to explicitly establish the way in which applications interact when they use common data. Finally, the environment may be used to define application oriented views of data, in a similar way to the view mechanism of DBMSs.

The examples, which are intentionally simple, concern the departments of a firm.

4.1. Data Persistence

Temporary values exist in the system only during the evaluation of the expressions in which they are defined. None of Galileo's data types defines permanent values. For instance, user programs may contain class definitions, if temporary classes are to be kept while running an application program. To deal with data persistence, a global environment exists in which all values are automatically maintained. The global environment is managed by the language support system. This approach to data persistence has been also described in (Albano 81, Atkinson 81).

When the user enter the system, he is in the global environment. New bindings are added with the construct "use EnvironmentExpression":

```
use GeneralManager := "Ada Byron"
```

Instead of having a single set of bindings, the user can fruitfully employ the environment mechanism to structure the global environment. For instance, in the following example the environment Personnel is defined modeling a database schema containing both classes and operations:

```
use Personnel :=
     (rec Departments class
           Department <->
            (Name : string
             and Manager : var Employee
             and Budget : var num)
             key (Name)
     and Emplcyees class
          Employee <->
           (Name : string
            and Sex:<Male or Female>
            and Salary : var num
            and Dept : var Department)
            key (Name));
     and ChangeDepartment :=
           function (d:Department,
                     e:Employee) is
              Dept of e <- d
     and EnrollEmployee :=
          function (n:string,
                    s:<Male or Female>,
                    sal:num,
                    d:string):
          Employee is
          mkEmployee
            (Name:=n
             and Sex:=s
             and Salary:=var s
             and Dept:=
                 var get Departments
                        with Name=d
                 if-fails
                 failwith "Unknown Dept.");
```

Each expression is evaluated inside an environment, initially the global one, called the current environment. Any environment that can be accessed from the global one can become the current environment with the command "enter environment". To return to the global environment there is the command "quit". For example, assuming that the classes in Personnel have already been populated, the following is a simple interactive session.

```
enter Personnel;
```

the definitions contained in Personnel become directly accessible and so, for instance, a new employee can be enrolled into the Research department with the function EnrollEmployee.

```
EnrollEmployee("Sibille Ellis",
               <Female>,
               1000,
               "Research");

quit;
```

4.2. Encapsulation

The environment mechanism can be used to model a schema as a set of interrelated units. Each unit encapsulates data and operations which are closely related. For instance, let us assume that we are interested in describing as distinct units data relevant to the Planning and Administration sectors of our hypothetical firm, although these sectors share data and functions of the environment Personnel:

```
use Planning :=
      (Personnel
       and Projects class
             Project <->
             (Name : string
              and Budget : var 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 the environment operators, the Personnel environment is shared by Planning and Administration, so that any updating of a class from an environment will be reflected in the others.

4.3. Refinements

New environments can be defined by extending other environments with new definitions. This possibility can be used both to personalize a schema with new data and operations and to refine a schema to generate a more detailed description of the database.

In the following example, an environment is defined as an extension of Personnel with the "PartTimeEmployees" class, which is a specialization of Employees. Thus data concerning the same application are visible at different levels of details.

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

       ext PartTimeEmployees
              subset of Employees class
            PartTimeEmployee <->
             is Employee
             and PrivateData : string;
```

"PartTimeEmployees" is a class subset of Employees which models an IS-A hierarchy; all PartTimeEmployees are Employees, but not vice versa. PartTimeEmployees must be populated explicitly by elements of the class Employees. Moreover PartTimeEmployee is defined as a subtype of Employee and so it inherits all the attributes of Employee, as well as having the additional attribute PrivateData. Consequently, because a subtype may be used wherever the supertype may be used, PartTimeEmployee can be used in any context where an Employee is expected. For example it is possible to apply the function EnrollEmployee to an element of PartTimeEmployees.

An important consequence of using environments together with the class specialization mechanism is that the general environment behaves really like a stable model of the application, and it can be refined later on when the model must be tailored to new requirements.

Environments can also be defined by combining more than one environment. For instance, the following ProjectManager environment is defined to include all data and operations of Planning and Administration, as well as its own data.

```
ProjectManager :=
  (Planning
   ext Administration
   ext Parts class
         Part <->
          (Code : int
        and Price : int
        and UsedBy : seq Project
        and SuppliedBy : Supplier)
        key (Code));
```

4.4. View Modeling

To provide filtered access to the database, it is possible to provide different views of an environment by excluding some of its data and operations.

```
use FemaleEmployees :=
    Personnel drop Employees;
```

In OnlyDepartments the Class Employees is not accessible. In the following environment, only female employees can be accessed and modified:

```
use FemaleEmployees :=
    (use Personnel
     in Women restriction of Employees
              with Sex is Female class
        Woman <-> is Employee);
```

In the previous example the class Women is a restriction of the class Employees contained in Personnel. Another way of modeling views is shown in the following example.

```
use MaleEmployees :=
   (use Personnel
    in Men :=
        derived
         for Employees with Sex is Male
           loop (Name:= Name
                 and Salary:= Salary
                 and Dept := Name of Dept));
```

Unlike the previous case, the sequence of elements bound to Men is calculated every time the identifier Men is used and cannot be updated. This way of modeling views is similar to virtual relations of relational DBMSs, in that the same set of operators is available to access virtual and real data.

In modeling views, it is also possible to change only the names of the objects in the schema:

```
use AnotherPlanning :=
   Planning rename Projects in ResearchProjects;
```

5. CONCLUSIONS

The problem of structuring complex database schemas has been discussed. Conceptual languages hitherto proposed support only the abstraction mechanisms of Semantic Data Models, which are based on the assumption that a data base should be modeled in terms of data abstractions explicitly intended to represent, naturally and directly, the semantics of the application. The abstration mechanisms have been shown to be an effective tool in schema design, but they do not help to structure a complex schema in smaller, related parts.

A solution has been presented centered around the conceptual language Galileo, in which this issue has been an important design goal. We believe that this paper provides evidence of how the environment construct allows a schema to be structured. This capability appears also to provide the linguistic support necessary to incrementally design databases, according to a methodology of stepwise refinement by specialization, proposed in the TAXIS project. Databases can be designed and tested incrementally, while preserving the levels of refinement, which can be used to give zoomed versions of an application: Different classes of users can then access the database at different levels of detail. It is interesting to point out that, with the proposed approach, a database is not seen as a monolithic entity, which can be accessed through views, but a database is modeled as a set of interrelated units.

ACKNOWLEDGEMENTS

We are indebted to the members of the Galileo Project for their constructive criticisms to the contents of the paper: L. Cardelli, F. Giannotti, B. Magnani, D. Pedreschi, and M.L. Sabatini. Also many thanks to M. Meyer for improving this paper's presentation.

REFERENCES

Abrial, J.R.,(74) "Data Semantics", in **Data Base Management**, J.W. Klimbie and K.L. Kofferman (eds), North-Holland Amsterdam, 1-60, 1974.

Albano A, M.E. Occhiuto and R. Orsini (81), "A Uniform Management of Temporary and Persistent Complex Data in High-Level Languages", in **Pergamon Infotech State of Art Report on Database**, M.P. Atkinson (ed.), Series 9, N.8, Pergamon Infotech Ltd, Maidenhead, 319-344, 1981.

Albano A, L. Cardelli and R. Orsini (83a), "Galileo: A Strongly Typed Interactive Conceptual Language", Technical Note N.30, University of Toronto, January 1983.

Albano A. and R. Orsini (83b), "Dialogo: An Interactive Environment For Conceptual Design In Galileo", in **Metodology and Tools for Database Design**, S. Ceri (ed.), North-Holland, Amsterdam, 229-253, 1983.

Albano A., (83c) "Type Hierarchies and Semantic Data Models", **ACM Sigplan '83: Symposium on Programming Languages Issues in Software Systems**, S. Francisco Ca, 178-186, 1983.

Amble, R., K. Bratbergsengen and D. Risnes, (79) "ASTRAL, a Structured and Unified Approach to Data Base Design and Manipulation", **IFIP Work, Conf. on Data Base Architecture,** Venice, Italy, 1979.

Atkinson M.P., K. Chisolm and E. I. Cockshott (81), "The New Edinburgh Persistent Algorithmic Language", in **Pergamon Infotech State of Art Report on Database,** M.P. Atkinson (ed.), Series 9, N.8, Pergamon Infotech Ltd., Maidenhead, 299-319, 1981.

Borgida A.T., J. Mylopoulos and H.K.T. Wong (82), "Methodological and Computer Aids for Interactive Information System Design", in **Automated Tools for Information System Design,** H.J. Schneider and A. Wasserman (eds), North-Holland, Amsterdam, 109-124, 1982.

Brodie M.L. (80), "The Application of Data Types to Database Semantic Integrity", **Information Systems 5,** 4, 287-296, 1980.

Brodie M.L. and S.N. Zilles (eds) (81), **Proc. Workshop on Data Abstraction, Data Bases and Conceptual Modelling, ACM SIGMOD Special Issue 11,** 2, 1981.

Capaccioli M. (83), "La Semantica Denotazionale del Galileo", Tesi di Laurea in Scienze dell'Informazione, Università di Pisa, Italy, 1983.

Codd E.F. (79), "Extending the Relational Model to Capture More Meaning", **ACM TODS 4,** 397-434, 1979.

Hammer M. and B. Berkowits (80), "DIAL: A Programming Language for Data Intensive Application", **Proc. of ACM SIGMOD Conference,** Santa Monica Ca, 75-92, 1980.

Hammer M. and D. McLeod (81), "Database Description with SDM: A Semantic Database Model", **ACM TODS 6,** 3, 351-386, 1981.

Hendrix, G. (75) "Expanding the Utility of Semantic Networks Through Partitioning", **Proceedings IJAI-75,** Tbilisi USSR, Sept. 1975.

McLeod D. and R. King (82), "Semantic Database Model's", in **Principle of Database Design,** S.B. Yao (ed), Prentice Hall, 1982 (to appear).

Montangero C., G. Pacini, M. Simi and F. Turini (78), "Information Management in Context Trees", **Acta Informatica 10,** 85-94, 1978.

Mylopoulos J. and H. Levesque (81) "An Overview of Knowledge Representation", **Proc. of the Workshop on Data Abstraction, Database and Conceptual Modelling; ACM SIGMOD Special Issue 11**,2,5-12, 1981.

Mylopoulos J., P.A. Bernstein and H.K.T. Wong (80), "A Language Facility for Designing Database-Intensive Application", **ACM TODS 5,** 2, 185-207, 1980.

Rowe L. A. and K.A. Shoens (79), "Data Abstraction, Views and Updates in RIGEL", **Proc. of ACM SIGMOD Conference**, Boston, Mass, 71-81, 1979.

Shopiro, J.E., (79) "A Programming Language for Relational Databases", **ACM TODS 4,** 4, 1979.

Smith J. M. and D.C.P. Smith (79) "Database

Abstraction: Aggregation and Generalization", **ACM TODS 2**, 105-133, 1979.

Smith J. M., S. Fox and T. Lancers (81), "Reference Manual for ADAPLEX", Technical Report CCA-81-02, Computer Corporation of America, January 1981.

Wasserman A. I. (79), "The Data Management Facilities of PLAIN", **Proc. of ACM SIGMOD Conference**, Boston, Mass., 60-70, 1979.

Wong H. K. T. and J. Mylopoulos (77), "Two Views of Data Semantics: A Survey of Data Models in Artificial Intelligence and Database Management", **Infor 15**, 3, 344-382, 1977.