A Formal Specification for a Database Model with Object-Oriented Features

Regina Motz [1]
Décio Fonseca

Departamento de Informática
Universidade Federal de Pernambuco
PO Box 7851, 50739 Recife - PE - Brasil

ABSTRACT: In this paper we present the formal specification of an object-oriented database model, the E/D Model. The E/D Model is an extension of the E-R Model, where structural and behavioral properties are treated explicitly in an integrated way. The E/D Model also possesses features of object-orientation: objects, object identity, types, classes and multiple inheritance. Object models have an important part to play in the future of database systems, but there is no agreement about exactly what an object-oriented database is, in spite of several attempts to provide a definition. In order to avoid semantic problems with terminology, we believe that formal specification is a good way of present our model. Besides that, the formal specification allow us to rapid prototyping the model. The Zc notation is used and briefly explained in order to understand this paper.

1 Why Formal Specification ?

There are good reasons to give formal description of a database model:

- The problems of producing reliable software quickly and economically have been widely discussed in recent years, and initial experience has indicated that the use of mathematical methods of specifying and designing software can contribute towards a solution.
- It provides precise, implementation independent, description of concepts.

[1]Present address: Resistencia 1636/102, CP 11400, Montevideo - Uruguay
e-mail: pardo@incouy.edu.ar

- A formal description may be regarded as a functional specification for the implementation.
- It provides a basis for reasoning about the *correctness* of programs -either directly, or by means of derived proof rules for correctness assertions.
- A formal description *documents* the design of the model.

On these approach of integration between Databases and Software Engineering the POESIS project [FS90] is been developed at the U.F.PE. The goal of the POESIS project is the construction of a multimedia environment for database modelling, implementation and access. The purpose is to provide a designer with an expressive tool for representing the information and to help a user to access this information. It provides a structured and cooperative menu-based interface [SFC90], an object-oriented management system (GOLGO) [MF89,Mac90,Sal89], and a knowledge base administrator. It is being developed a first data model, called E/D (an acronym for the words static/dynamic in Portuguese) [Ban89,Mot90], which is also being formally specified using the Zc notation. A prototype of the E/D Model has been implemented using Smalltalk/V. This prototype currently provides two data operations: schema defi nition and database creation.

2 An Overview of the E/D Database Model

This section introduces the basic concepts which the E/D Data Model deals with. The E/D Model is an extension of the E-R Model, where structural and behavioral properties are treated explicitly in an integrated way. The E/D Model also possesses features of object-oriented data models: objects (everything instance is an object) , object identity (objects have an unique identifier), types (same as abstract data type concept in programming languages), classes (agruoping objects) and multiple inheritance.

An *object* is a real-word element or concept which can be distinctly identified. The primary characteristic of an object is that it has an identity which persists through time, however the properties of the object may change. This identity may be considered to be represented by a system generated object identifier (oid). Objects may be explicitly created and destroyed.

Every object is an instance of one or more classes and every classe has an associated type. There is the usual notion of a type lattice, i.e. the possibility of multiple inheritance. The relation among the types is the common database relation of 'IsA' and the relation among the classes is the 'SubSetOf' relation (Figure 1).

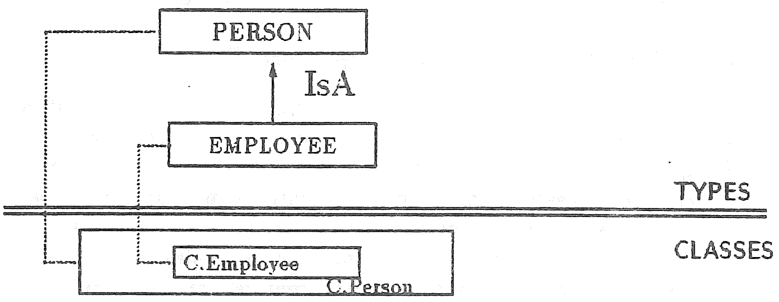

Figure 1: Relations of "IsA" and "SubsetOf".

A type can have multiple supertypes as graphically shown Figure 2.

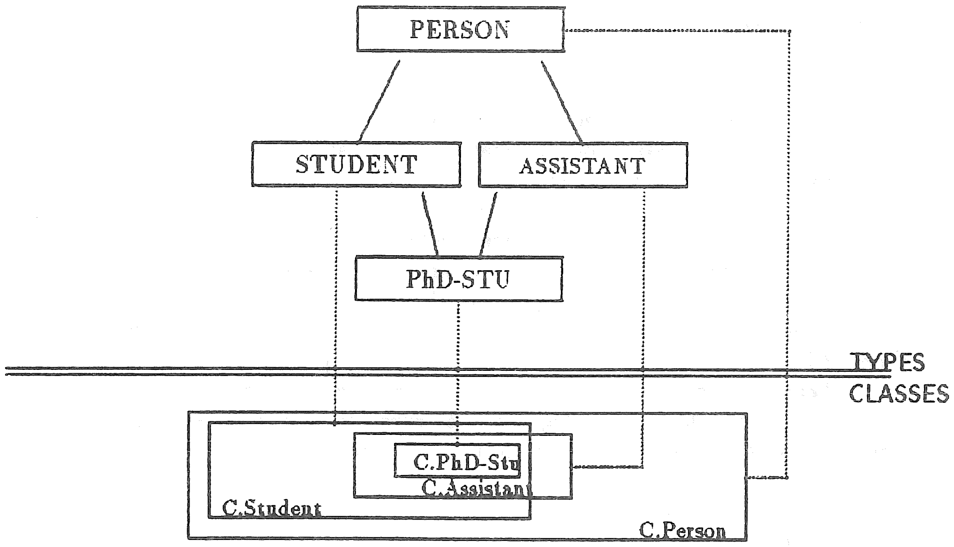

Figure 2: Multiple inheritance among types and classes.

The user must resolve conflicts between inherited structures or methods by writing a new structure or method, or explicitly inherit one instance variable or method from among several conflicting ones.

3 A Formal Description of the E/D Model

3.1 The Specification Language Zc

The Zc notation, a language for expressing mathematical specifications of complex computing systems, is been developed by th Software Engineering Group at UFPE [Sam88,Sou88]. It is based on the Z notation [Suf86], and as well as Z, is based on first order predicate calculus and typed set theory.

Zc increases the notion of *schema*, existing in Z, with the notions of *chapters*, *documents* and *library*, which improve its modularity and reusability.

A *schema* is a named piece of mathematical text that introduces some variables, and some predicates which express a constraint relating the possible values of the variables. Schemas are given in the form:

```
> SCHEMA Schema_S;
>    VAR
>       variables
> PREDICATE
>       predicates;
> END Schema_S;
```

Schemas can describe operations on data states. Some schemas require input and/or output data. Conventionally we decorate inputs with a final query, and outputs with a final shriek.

Although the style in which schemas are used in Zc is quite different from that used in Z, both languages have facilities for combining these structural units, and both make some provision for parametrized specification units.

In Z schemas are used for everything whereas in Zc there are specific syntatic units for defining functions (DEF), types (TYPE) and algebraic types (ALGTYPE).

A *chapter* clusters types, definitions, schemas and theorems, whereas a *document* groups reference to chapters and represents the whole specification of the system. Documents are given in the form:

```
> DOCUMENT DocName;
>       ChaptersNames;
> END DocName;
```

A *library* contains documents and chapters. Specifications of large systems are built up by specifying smaller sub-systems using chapters, then grouping these chapters into a document. Zc allows the importation and exportation of objects[2] among the chapters and the definition of types and local definitions into a schema.

The documentation used is the one known as *literal programming*: lines beginning with the symbol > are lines of specification. The advantage of this type of documentation is that it allows to coexist in a single document the comments and the specification.

3.2 The Types of the E/D Model

In the E/D Data Model the type of an object may be system-defined or user-defined, and there is no disctintion between them. The system-defined types are: the atomic types (Integer, Real, String, Bool and Und[3]), the types constructs (Set, List and Record); and the predefined types of the E/D Model (UserT, TConstr, Entity, Rela, Action, Rule, Trans and MetaRule).

```
> ALGTYPE
>       Type :: user UserT          // User defined type
>             | restr TConstr       // Constraint
>             | ent Entity          // Entity
>             | rela Rela           // Relationship
>             | ac Action           // Action
>             | ru Rule             // Rule
>             | tran Trans          // Transaction
>             | metaru MetaRule;    // MetaRule
```

Types are organized in a type structure that supports generalization and specialization, this structure is a lattice. A type may be declared to be the subtype of another type.

Every type has a common information of it: a description. its supertypes, the excludents types with it and a set of operations.

```
> SCHEMA TypeInf;
>   VAR
>       description   : Message,
>       supertypes    : SET TypeName,
>       excludent     : SET TypeName,
>       operations    : Ops;
> END TypeInf;
```

[3]In this Section, the term *object* is used to reference to types, definitions, schemas and theorems.

[3]UND–The object undefined. This type is obtained every time an operation is not applicable.

The structure of a type is modelling as a parametrized record, it depends of the specific type. The representation of the structure as a record allows us to eficiently management the multiple inheritance.

```
> TYPE Structure[X] = MAP Label TO X;
```

3.2.1 Static Features

In this subsection we present the specification of the static features of the E/D Model: UserT, TConstr, Entity and Rela.

$\boxed{\text{UserT}}$

The user-defined types permit the extensibility of the model. Because the diversity of types required by the new applications, with multimedia documents, it is insufficient to extend the database system to include a specific and limited set of data types. For these reason the object-oriented approach allows the user to define methods on types in an incremental way. Our approch is similar in the sense that it lets the user define the data types and methods which are specific to the application area.

```
> SCHEMA UserT;
>    INCLUDE TypeInf.
>    VAR
>          struct      : Structure[TypeConstr],
>          constraints: SET NameTConstr;
> END UsuarT;
```

```
> ALGTYPE TypeConstr ::   ac AtomComp
>                        | set-of TypeConstr
>                        | seq-of TypeConstr
>                        | record Structure[TypeConstr];
```

```
> ALGTYPE AtomComp :: atomname Atom | typename TypeName;
>             Atom = Integer | String | Real | Bool;
```

$\boxed{\text{TConstr}}$

The type TConstr specify the constraints of the model. Exist four predefined constraints, which indicates that the values in the indicated labels of every object of a type with one of these constraints must be:

- defined for the *exist* one
- not null for the *not-null* one
- unique in the class for the *unique* one
- verify the condition for the *condition* one

```
> SCHEMA TConstr;
>   INCLUDE TypeInf.
>   VAR
>         struct: Structure[Constr]
> END TRestr;

> ALGTYPE Constr ::  exist (SET Label)
>                   | not-null (SET Label)
>                   | unique (SET Label)
>                   | dinamic Condition;
```

Entity

An entity concept has a name, a description, its parents, a structure, a set of restrictions and the set of total relationships in which the entity is a member. Moreover, each entity type in the database may posses at least one field in its structure then it has the following invariant: *the cardinality of the domain of its structure must be greater/equal than one.*

```
> SCHEMA Entity;
>   INCLUDE TypeInf.
>   VAR
>         struct      : Estrutura[AtomComp],
>         constraints: SET NameTConstr,
>         Totalrela   : SET RelaName;
> PREDICATE
>         CARD DOM struct >= 1;
> END Entity;
```

Relationship

Relationships have sets of entities associated with them, indicating the participating entities, as depicted in the E-R diagrams. Role functions allow to trace back the participating entities. Roles are connections between entities. E.g. the *ChildOf* relationship has the two roles *child* and *parent* that both must hold identifications to entities of the *person* type. Each role possesses an associated cardinality, which indicates the minimum and maximum times that an specific entity can participate in that role.

```
> TYPE Roles = MAP OpName TO Cardins;


> TYPE Cardins = (cardmin: INTEGER,
>                         cardmax: Cardin),
>          | (FORALL c: Cardins •
>          c.cardmin < c.cardmax AND c.cardmin >= 0);


> ALGTYPE Cardin = 'M' | int INTEGER;
```

It is possible to define constraints on the relationship's structure or on the participating entities. then exist the followinf type NameConstrRela in the specification:

```
> TYPE NameConstrRela ::   on-rela NameTConstr
>                          | on-partic (NameTConstr, Label);
```

A relationship concept has a name, a description, a set of entities partipating. a structure and a set of connections having the same role. It has also the following invariant: *each relationship in the database may be a relation between at least one entity type and there must be at least two roles.*

```
> SCHEMA Rela:
>    INCLUDE TypeInf.
>    VAR
>          partic      : SET NameEntity,
>          roles       : Roles ,
>          struct      : Structure[AtomComp],
>          total       : BOOL.
>          constraints : SET NameConstrRela.
> PREDICATE
           The relationship must have at least one participating entities
>          CARD partic >= 1 AND

           At least two roles must be defined
>          CARD DOM roles >= 2 AND

           The structure of the relationship includes the participating entities
>          partic ⊆ RNG struct AND

           The roles are defined among participating entities
>          FORALL opname:OpName | opname IN DOM roles •
>          Elems (operations opname).op-type ⊆ partic AND
```

```
            The constraints must be defined among existing labels in the
            relationship's structure or on the participating entities
>           FORALL c: NameConstrRela •
>           EXIST nc: NameTConstr, lab: Label | c IN constraints AND
>                     c = on-partic (nc, lab) •
>                     lab IN DOM struct AND (struct lab) IN partic;
> END Rela;
```

3.2.2 Dynamic Features

In this subsection we present the dynamic features of the E/D Model: Action, Rule, Transaction and Metarule.

$\boxed{\text{Action}}$

The action type allows the user to define its own conceptual operations over entities and relationships types. These conceptual operations use a set of primitives such as Insert, Delete, Modify, Let, and others pre-defined actions.

```
> SCHEMA Action;
>   INCLUDE TypeInf,
>   VAR
>         struct: Structure[ActionBody],
> END Action;

> TYPE ActionBody = (on : SET NameEntRela,
>                    act: SET OpActT);
```

$\boxed{\text{Rule}}$

The Rule type represents the behavioral properties of the entities and relationship types. It is a trigger in the model and can be seen as a conditional action.

```
> SCHEMA Rule;
>   INCLUDE TypeInf.
>   VAR
>         struct: Structure[RuleBody];
> END Rule;

> TYPE RuleBody = (action : NameAction, cond : Cond);

> ALGTYPE Cond :: cond-op CondOper | verif Condition;
> TYPE CondOper =(if : NameCondition, next-action : NameAction);
```

Transaction

The Transaction type groups actions and rules that modifies the state of one or more objects, which must be executed in a defined order and completed.

```
> SCHEMA Trans;
>    INCLUDE TypeInf,
>    VAR
>           struct: Structure[TransBody];
> END Trans;

> TYPE TransBody = (on  : SET NameEntRela,
>                   act : SEQ NameAcRule);
```

Metarule

The Meta-rule type allows the definition of priorities among the execution of several rules and controls the execution of exclusive ones.

```
> SCHEMA MetaRule;
>    VAR
>           struct: Structure[MetaRuleBody],
> END MetaRule;

> TYPE MetaRuleBody = (on             : SET NameEntRela,
>                      exclu-priority : SET (SEQ NameRule));
```

3.3 The E/D Instance and Scheme

The E/D Model consists of two part: the Instance and the Scheme.

Instance

An Instance of the E/D Model consists of a partial function (ObjMem) which associates an object for each object identifier.

```
> TYPE ObjMem = MAP Oid TO Object;
```

Each object has a state and an associated type (which is the most specialized type of it). These components of an object must verify an invariant which says that the structure of the state and the type must be in correspondence.

```
> TYPE Object = (sp-type: TypeConstr,
>                state:    ObjState);
>        | FORALL obj: Object •
>           (EXIST a: Atom | obj.sp-type = ac (atomname a) •
>           EXIST v: AtomicValue • obj.state = atomic v)
>        AND
>           (EXIST tc: TypeConstr | obj.sp-type = set-of tc •
>           EXIST s: SET Oid • obj.state = set s)
>        AND
>           (EXIST tc: TypeConstr | obj.sp-type = seq-of tc •
>           EXIST s: SEQ Oid • obj.state = sequence s)
>        AND
>           (EXIST n: TypeName | obj.sp-type = ac (typename n) •
>           EXIST m: MAP Label TO Oid •
>           obj.state = record m);
```

The structure of the object state (ObjState) shows that all the *type instances* has an uniformity representation as "objects".

```
> ALGTYPE ObjState = atomic AtomicValue |
>                    | set (SET Oid)
>                    | sequence (SEQ Oid)
>                    record (MAP Label TO Oid);

> ALGTYPE AtomicValue =  integer INT
>                       | real REAL
>                       | string STRING
>                       | bool BOOL
>                       | undef UNDEF;
```

$\boxed{\text{Scheme}}$

A Scheme contents the information about: Types and Classes.

```
> SCHEMA SchemeE/D;
> VAR
>       tenv     : TypeEnv,
>       classes  : Classes;
> END SchemeE/D;
```

The type environment associates a type for each type name. It contains all the types defined in an application. Each type in the environment must verify that its supertypes or excludent types also exists in the environment, and that the excludent constraint is not violated.

```
> TYPE TypeEnv = MAP TypeName TO Type
>          | (FORALL Tenv: TypeEnv, t: Type |
>                 t IN RNG Tenv •
>                 t.supertypes ⊆ DOM Tenv AND
>                 t.excludent ⊆ DOM Tenv AND
>                 NOT EXIST st:TypeName | st IN t.supertypes •
>                 (Tenv st).excludent ∩ t.supertypes <> {});
```

Each class has an associated type (which indicates the type of its objects) and an extension (which groups its objects).

```
> TYPE Classes = MAP ClassName TO ClassBody;
>          ClassBody = (type  : NameEntRelaUser,
>                       ext   : SET Oid);
```

DataBase

A DataBase consists of an instance (ObjMem) and a scheme, such that every object of the ObjMem belongs to a unique class of the scheme.

```
> SCHEMA DataBaseE/D;
> INCLUDE SchemeE/D
> VAR
>       objects: ObjMem;
> PREDICATE
>       FORALL id: Oid | id IN DOM objects •
>       UNIQUE c: ClassName | id IN (classes c).ext
> END DataBaseE/D;
```

3.4 Primitive Operations

The following are the primitive operations that are implemented in the current prototype of the E/D Model.

- Inicialization
- Create Type
- Generalization of a Type
- Insert operations in a Type
- Navegation among the Types
- Delete Type
- Create Class

- Delete Class
- Create Object
- Delete Object

At following we show the specification of the operation of create an user-defined type.

3.4.1 An Example: Create an user-defined type

```
> SCHEMA NewUserT;
>    INCLUDE NewInfType,
>    VAR
>          fields?   : SET Field[TypeConstr],
>          constr?   : SET NomeTConstr,
>          struct    : Structure[TypeConstr],
>          st        : Structure[TypeConstr],
>          t!        : UserT;
> PREDICATE
           The new type must be a valid type in the environment
>          PRED-ValidType AND

           The new type is generated
>          t!.description = message? AND
>          t!.supertypes = supertypes? AND
>          t!.excludents = excludents? AND
>          t!.struct = struct AND
>          t!.constraints = constr? AND
>          t!.operations = Make-Ops operations? AND

           The type environment is updated
>          Tenv' = Tenv $ {name? ↦ t!};
> END NewUserT;


> SCHEMA ValidType;
>    INCLUDE NewTypeInf,
>    VAR
>          fields?   : SET Field[TypeConstr],
>          constr?   : SET NameTConstr,
>          st1, st2  : Structure[TypeConstr]
> PREDICATE
           The fields of the structure of the new type
           are valid types in the environment.
>          VerifiySet (ValidField Tenv) fields? AND

           The structure of the type is built.
>          st1 = Make-Structure fields? AND
```

```
            The constraints must exists in the type environment.
>           constr? ⊆ DOM Tenv AND

            The structure of the type is a valid refinement of its supertypes.
>           VerifySet (Refine Tenv st1) supertypes? AND

            There is not clash among the multiples supertypes.
>           PRED-MultInherit AND

            The structure of the type with its supertypes is built.
>           st2 = BuildTypeStruct Tenv name? st1 AND

            The constraints are defined on existing labels of the type.
>           VerifySet (ValidConstr Tenv st2) restric?
> END ValidType;

> SCHEMA MultInherit;
>    INCLUDE SchmeE/D;
>    INCLUDE CommomInherit;
>    VAR
>           supertypes? : SET TypeName,
>           st?         : Structure[TypeConstr],
>           ssl         : SET(SET Label)
> PREDICATE
            Without multiples supertypes
>           CARD supertypes? =< 1 OR

            The set of labels of the type and its supertypes structure is built.
>           ssl = mapset (Labels Tenv) supertypes? AND

>           (CARD supertypes? > 1 ⇒
            Exists multiples supertypes without clash.
>         Clashes ssl = {} OR

            Exists clashes in multiples supertypes that may be resolved if the common label
            are redefined in the new structure of the type or
            if they belong to a common supertype.
>         Clashes ssl <> {} ⇒Clashes ssl ⊆ (DOM st? ∪ labs!))
> END MultInherit;
```

4 Conclusion

In our work, first of all we provided a brief review of the basic object-oriented concepts extracted from existing object-oriented systems in order to form the basis of our data model. We decided to emphasize on the notion of type. In database applications, the ability to add new data types allows one to model more easily and precisely a given application domain. User-defined domains may be described in the E/D Model, which makes possible the inheritance of state variables and methods among domains, allowing the easy extensibility of the model.

We elaborated a formalization of our model using the specification language Zc. Zc was an adequate tool for the specification, especially by the fact that it provides a module structure. Chapter allows us to factor out common parts of a specification as can be seen in [Mot90]. This give us the opportunity to subordinate complexity. Formal specifications as the one shown in this paper can be a basis for direct prototyping. In our case, we obtained an object-oriented prototype in Smalltalk/V [Gol83] from the total specification of the E/D Model (shown in [Mot90]) using the method described in [MTM90]. With the help of this method the development of the prototype has been fairly easy and allowed in some sense the verification of the formal specification.

Last, but not least, we are exploring the integration of different programming languages with user-oriented caracteristics in order to help users to control their queries. The next step in order to achieve our objective is to develop a query language with object-oriented features in a formal way, i.e. give to it a formal semantics which will use this formal specification of the E/D Model as its semantic domains.

References

[Ban89] M. Bandeira. *Modelagem Estática/Dinâmica de Sistemas de Informações.* Master's thesis, Universidade Federal de Pernambuco, Recife, 1989.

[FS90] D. Fonseca and A. C. Salgado. Ambiente Integrado de Desenvolvimento de Banco de Dados Multimídia. In *Anais do XV Congresso Latino Americano de Informática*, CLEI, Asunción - Paraguay, 1990.

[Gol83] A. Goldberg. *Smalltalk-80: The Language and its Implementation.* Addison-Wesley, 1983.

[Mac90] M. A. C. Macedo. *GOLGO: Gerenciador orientado a objetos.* Master's thesis, Universidade Federal de Pernambuco, Recife, 1990.

[MF89] M. A. C. Macedo and D. Fonseca. GOLGO: Gerenciador orientado a objetos. In *Anais do IV SBBD*, Simpósio Brasileiro de Banco de Dados, Campinas, 1989.

[Mot90] R. Motz. *Estudo Formal de um Modelo de Dados Orientado a Objetos.* Master's thesis, Universidade Federal de Pernambuco, Recife, 1990.

[MTM90] R. Motz, F. Tepedino, and S. Meira. From model based specifications to object-oriented prototypes. In *Anais do X Congresso da SBC*, Sociedade Brasileira de Computação, Vitória, 1990.

[Sal89] A. C. Salgado. O nível físico orientado a objetos de um SGBD multi-midia. In *Anais do IV SBBD*, Simpósio Brasileiro de Banco de Dados, Campinas, 1989.

[Sam88] A. Sampaio. *Zc: Uma Notação para Especificação de Sistemas Complexos.* Master's thesis, Universidade Federal de Pernambuco, Recife, 1988.

[SFC90] A. C. Salgado, D. Fonseca, and M. Casado. Umiformização de interfaces de comunicação em ambiente multimídia. In *Anais do V SBBD*. Simpósio Brasileiro de Banco de Dados. Rio de Janeiro, 1990.

[Sou88] R. Souto. *Especificação Formal de Software de Grande Porte - Um Exemplo Real.* Master's thesis, Universidade Federal de Pernambuco, Recife, 1988.

[Suf86] B. Sufrin. *The Z Handbook.* Programming Research Group, Oxford, 1986.