CONCEPTUAL LAYERS IN AN OBJECT ORIENTED DATABASE: AN UNIFIED MODEL

Prof. Gabriel Baum
Lic. Claudia Pons
Carlos Tau

Prof. Silvia Gordillo
Clara Smith

LIFIA, Departamento de Informática, Facultad de Ciencias Exactas,
Universidad Nacional de La Plata. Calle 50 esq. 115. 1er piso.
Tel: (54)(21)4-2738. E-mail: lifia@cespivm2.unlp.edu.ar
Fax: (54)(21)25-6607. (54)(21)24-4300.
(1900) La Plata, Bs. As., Argentina.

ABSTRACT

When the programming languages for database applications emerged, a greater abstraction for representing knowledge was reached. Rapidly, specific application areas needed a precise conceptual modeling of the data. As an object oriented paradigm allows modeling real-world situations in a clear and natural way, it seems interesting to combine its principles with those of database development.

The sound and reusable development of Object Oriented Databases (OODB) should be possible only on the top of a strong theoretical basis. This work presents an object oriented approach to the specification and design of an OODB, based on the following key concepts: objects, abstraction mechanisms for defining object types, relationships among objects, object composition, and multiple implementations. It also researches sound theoretical foundations for building OODB in a correct, verifiable, and reusable way. The proposal includes: a) a conceptual model of an OODB, b) an algebraic specification language for defining OODB elements.

INTRODUCTION

An object is a dynamic entity: it has an internal state that may be modified in response to certain events, and can only be partially observed through its features. An object has also the power of performing certain activities (its behavior). Far from being isolated entities, objects organize themselves in different ways, and coexist in a complex and dynamic society, acting on each other. When the programming languages for database applications emerged, a greater abstraction for representing knowledge was reached.

The OODBs provide facilities for manipulating abstraction mechanisms (such as classification, generalization-specialization, composition, and parameterization) and for capturing connections and restrictions among objects. The OODB systems ORION [4] and IRIS [3], the O2 language and its extensions for database management [6], and the extensible system for databases EXODUS [2] were studied.

O2 proposes a separation between a type specification and its corresponding implementation. It distinguishes private from public methods, and allows the creation of exceptional objects. EXODUS separates the type definition from the instances declaration, and allows the manipulation of several sets of objects belonging to the same type. Also it offers an interesting definition of component objects. IRIS introduces the possibility of defining disjoint types. ORION supports the management of invariant rules for schema manipulation and presents the notion of composite object through the "is_part_of" relationship.

These four systems constitute the incentive for working on a unified model for OODBs.

This paper is organized as follows: section 1 presents a model for OODBs, consisting in three conceptual layers. Section 2 describes formal syntax and semantics for the elements belonging to each layer. Finally, a case of study is encountered in section 3.

1. STRUCTURE OF AN OODB

In an OODB three different planes, layers or levels can be identified: the definition layer, the schema construction layer and the implementation layer (Fig.1).
The definition layer consists in specifying object types. This process originates a type universe, with types organized in a hierarchy and with eventual composite references between them.

In the schema construction layer, those types of interest (for the application) are selected. These types are well fitted for the application needs. Also relationships between the selected types are defined in this layer.

The third layer reflects the construction of implementations for the types in the universe. It is possible to define multiple implementations for each type, building a hierarchy with them.

Extra details on this layer are not found in this paper.

1.1 Definition Layer

1.1.1 Abstraction mechanisms for building the universe.

An important concept in object orientation is abstraction, as a powerful tool for managing complexity. On constructing the universe, three useful abstraction mechanisms are provided: classification, composition and parameterization.

a- To classify consists in grouping similar objects in object types. A type specifies the common properties of its instances.

Also, types can be organized hierarchically (according to their abstraction laws) in such a way that a type in the hierarchy is an abstraction of all its descendent types ("is a" relationship), providing the generalization-specialization mechanism.

b- Composing means clustering objects in another object, frequently named composite object, associating properties that govern the interaction and behave as a whole. The typical relationship that occurs in a composite object is the "is part of" relationship [5].

c- Parameterization is a mechanism that gives the possibility of defining generic types. Generic types play the role of patterns, receiving types as arguments and generating new types.

According to the abstraction mechanisms presented, a type can be:
- a basic type,
- an extension of an existent type ("is a" relationship),
- a composition of existent types ("is part of" relationship).

Inside the definition layer, two parallel sub-layers can be distinguished:
- the "concrete" sub-layer, which contains type specifications, and
- the "generic" sub-layer, formed by parameterized type definitions that can be considered functions ranging over the concrete sub-layer.

1.1.2 Features and behavior.

Type operations can be classified in two well differentiated groups: features and behavior.

Features define characteristic and descriptive parts of the objects of a specified type; the features are the operations that permit the observation of the object's state. Each feature recovers some descriptive attribute (either showing the contents of some structure used to represent the type, or evaluating some function whose domain is the object itself).

The behavior is the set of activities that objects belonging to the specified type are capable of performing.
1.2 Schema Construction Layer

When defining the schema for the database, those types (specified in the definition layer) which reflect the real world entities features are selected from the universe.

Relationships among the selected types are defined. Some "is-part-of" relationships in composite objects may be fitted, as well.

Application restrictions are established in this layer as schema axioms.

Every definition made during the schema construction is valid only in the current schema; changes and adjustments have no effect upon the definition layer.

1.2.1 Relationships among types

Although relationships are elevated to the same abstraction level as types (instead of being stated inside of them), a relationship is modeled in a different way from a type: the former describes a set of groups of associated objects (the object oriented view of a table), while the latter defines characteristic features of an entity.

Relationships can be derived from existing generic specifications (which belong to the generic sub-layer), such as Binary Relation, Partial Order Relation, and so on.

Defining features and behavior when setting up a relationship among types, allows representing its descriptive attributes.

1.2.2 Structured objects

Four kinds of composite links among objects are presented, each with two versions. These links represent different "is-part-of" references between objects:

i- Exclusive component of a type: an object o of type O is a component part of another object t of type T, and cannot be (at the same time) a component part of any other object with type different from T. This kind of reference reflects a "logical component".

Version a) o can exist in the database, no matter t, ..., tn exist or not.

Version b) o's existence is conditioned to the existence of at least one ti in the database.

ii- Homogeneously shared component: an object o of type O is a component part of various objects, say t1, ..., tn of type T, but cannot be (at the same time) a component part of any other object with type different from T. This kind of reference reflects a "logical component".

Version a) o can exist in the database, no matter t1, ..., tn exist or not.

Version b) o's existence is conditioned to the existence of at least one of the objects t1, t2, ..., tn.

iii- Heterogeneously shared component: an object o of type O is a component part of an object t of type T, and may also be a component part of various objects (say a, b, c, ...) of any type different from T.

Version a) o can exist in the database, no matter r, s, t, ... exist or not.

Version b) o's existence depends on the existence of at least one of the objects r, s, t, ...

iv- Freely shared component: an object o of type O is a component part of various objects, say r, s, t, u, ..., no matter what type they belong to.

Version a) o can exist in the database, no matter r, s, t, u, ... exist or not.

Version b) o's existence is conditioned to the existence of at least one of r, s, t, u, ...

This classification is quite wide to cover a large set of object composition alternatives. In addition to these references, weak (standard) references between objects should be supported.

Restrictions imposed to a type T, when composing objects, are suitable for the subtypes of T, preserving the "is_a" relationship semantics.

1.3 Implementation Layer

It is widely accepted that separating the specification from the implementation is a good design mechanism. On the one hand, specification establishes features and behavior, as well as classification and composition of types. On the other hand, a type implementation provides representation and methods associated with features and behavior specified.

In the implementation layer, implementations for the types conforming the universe are build. Multiple implementations of a type are supported. A multiple implementation is a tree, where each node is an implementation of its parent. Nodes in the tree represent different levels of abstraction, beginning with the root (which is the most abstract level) and ending in the leaves, which are the concrete implementations [8].
The fact of dealing with several implementation alternatives for a type, allows the selection of the best choice for a particular situation, increasing in this sense efficiency and reusability.

2. THE SIGMA DEFINITION LANGUAGE

The design of suitable data structures is one of the crucial points in the construction of large software systems. Inner composition, properties and available operations of these data structures determine questions of simple realizability on concrete systems, provability of correctness and simple handling by users. One of the most important developments in the area of software technology are the 'Abstract Data Types', where data structures are characterized by the sorts (names of their carrier sets), the names of the operations and their properties; most of the concrete implementation details, which may cause problems for the understanding and the portability, can be omitted that way.

An abstract data type can be seen as a class of total heterogeneous algebras. That type can be specified by a signature (a collection of sorts and operations) together with a set of axioms [7].

ASL is a well-typed higher order language for algebraic specifications. It contains constructions for building signatures, sets of terms, and sets of formulas as well as constructs embodying primitive operations on algebraic specifications. It also allows the construction of parameterized specifications [7].

In this work is presented an algebraic specification language for defining OODBs, called SIGMA. This language is based on the ASL language. SIGMA provides constructors for formally specifying the elements (eventually generic) belonging to each conceptual layer in an OODB:

i- Definition layer:
- Basic type constructor:
  Takes a signature and a set of axioms, and creates a type specification.
- Subtype constructor:
  Enriches (or extends) an existent type specification, forming a type specification.
- Composite type constructor:
  Makes a type specification composing existent type specifications (the constituent parts), allowing the association of properties in order to define the whole.

ii- Schema construction layer:
- Relationship constructor:
  Permits the establishment of a relation among two or more types (the components).
- Constructor for fitting composite relationships:
  Sets up the kind of composition reference (exclusive, shared, dependent, independent) in an established 'is part of' relationship between objects.
- Schema constructor:
  The schema definition includes: the relevant types selection, the definition or relationships among the selected types, and the establishment of application-sensitive restrictions.

2.1 Syntax

i) SIGMA expressions belonging to the definition layer:
<def_expr>::=<basic_spec>|<subtype_spec>|<comp_spec>

<basic_spec>::=Btype<expr>
Features:<expr>
Behavior:<expr>
Axioms:<expr>
endtype

<subtype_spec>::=Stype<expr>
Supertypes:<expr>
Features:<expr>
Behavior:<expr>
Axioms:<expr>
endtype

<comp_spec>::=Ctype<expr>
Uses:<expr>
Features:<expr>
Behavior:<expr>
Axioms:<expr>
endtype

ii) SIGMA expressions belonging to the schema construction layer:
<schema_const_expr>::=<rel_spec>|<fitting_spec>|<schema_spec>

<rel_spec>::=Relation<expr>
Components:<expr>
Features:<expr>
Behavior:<expr>
Axioms:<expr>
endrel
<fitting_spec> ::= 
customize type <expr>
with
  (<expr> : <kind_link>)*
oncustom

<kind_link> ::= <dep indep> <link>
<dep indep> ::= dependent | independent
<link> ::= exclusive | homogeneously shared | heterogeneously shared | freely shared

<schema_spec> ::= 
  Schema <expr>
  with
    Types: <expr>
    Relations: <expr>
    Axioms: <expr>
  endschema

2.2 Semantics

For the understanding of the semantic of ASL-specifications, every specification can be seen as a pseudo-axiomatic class of algebras, that is, a class of algebras which can be axiomatized using hidden sorts operators; and every parameterized specification can be seen as a function from specification to specification [7].

Every SIGMA expression can be translated into an equivalent ASL expression. This means SIGMA's semantics can be expressed in terms of ASL's semantics.

In order to simplify the translation of SIGMA expressions, the following abbreviation is used:

enrich T by sorts S opns F axioms E = T + (signature <(sorts sig T) U S, (ops sig T) U F>, axioms E)

Let spec be a function that associates a specification name with its corresponding text.

i) SIGMA expressions belonging to the definition layer:

a) A <basic_spec> expression is directly translated into a <basic_spec> ASL expression.

Btype T
  Features: F
  Behavior: B
  Axioms: E
endtype =
  signature <T, F U B> axioms E

b) A <subtype_spec> expression involves the renaming of the sort of interest (inheritance) and an eventual enrichment with operations and axioms.

Ssubtype T
  Supertypes: S
  Features: F
  Behavior: B
  Axioms: E
endtype =
  enrich (derive from spec S by [S -> T])
    by opns F U B
    axioms E

This notion of subtype settles an inclusion relationship between the models for the types of interest. Given a type T, T subtype of S, every model for T, restricted to S's signature, is also a model for S.

c) A <component_spec> expression can be translated as the sum of all the component sorts, subsequently enriched with new sorts, operations and axioms.

Ctype T
  Uses: T1, ..., Tn
  Features: F
  Behavior: B
  Axioms: E
endtype =
  enrich(spec T1+...+spec Tn)
    by sorts T
    opns F U B
    axioms E

ii) SIGMA expressions belonging to the schema construction layer:

The central concept in this layer is the definition of relationships among types.

a) Semantically speaking, relations behave the same way as types.

Relation L
  Components: T1, ..., Tn
  Features: F
  Behavior: B
  Axioms: E
endrel =
  enrich(spec T1+...+spec Tn)
    by sorts L
    opns F U B
    axioms E

b) The BDOO schema specification is build up from the sum of its components' specifications and from the attachment of application-specific axioms. The BDOO schema semantics can be understood as the set of all the valid BDOO instances of that schema.
Schema S
with
  Types: Tl,..,Tn
  Relations: Rl,..,Rk
  Axioms: E
endschema = enrich(spec Tl+...+spec Tn +
  spec Rl +...+ spec Rk )
  by axioms E

3. A CASE OF STUDY

3.1 A Type Universe Construction.

Considering a situation where a Football Association organizes an annual Championship, in which several teams take part, the following are types representing persons and football players. Both types belong to the type universe:

Ctype Person
  Uses: Integer, String, Boolean
  Features:
    name: String
    age: Integer
  Behavior:
    birthday: Person -> Person
  Axioms:
    age(birthday(P)) = age(P) + 1
  endtype.

Stype Football-Player
  Supertypes: Person
  Features:
    position: Integer
    goals: Integer
  Behavior:
    make_goal: Football-Player -> Football-Player
    ...
  Axioms:
    goals(make_goal(J)) = goals(J) + 1
  endtype.

A football team will be modeled as a set of football players. The generic specification of a Set type will be used to generate the concrete type Footballers-Set.

Footballers_Set = Set(Football_player)

Football players, together with their coach, form a Football Team. The strategy used by the coach to train the footballers is a private feature, and cannot be observed from outside a football team object.

Ctype Football_Team
  Uses:
    Footballers_Set, Person
  Features:
    name: String
    coach: Person
    football_players: Footballers_Set
    strategy: String Private
  Behavior:
    replace_coach: Football_Team, Person
    -> Football_Team
    replace_strategy: Football_Team, String
    -> Football_Team
  Axioms:
    coach(replace_coach(T, P)) = P
  endtype.

3.2 Relationship Specification.

A match could be modeled as a pair of football teams. A championship could be thought as several match-days. During each match-day, every team disputes a match with another team.

The following is a specification for the relationship Match-date:

relation Match-date
  Components: Football_Team, Football_Team
  Features:
    score: Integer
    referee: Person
  Behavior:
    addMatch: Match-date, Football_Team, Football_Team
    asignReferee: Match-date, Football_Team, Football_Team, Person
  Axioms:
    ...
  endrelation.

Relationship definitions are also suitable for the subtypes of the types related. For example, if the type Boy belongs to the universe, a boy can be a member of a football team. But some axioms may restrict the possibility of conforming a football team to persons over 18 years. Relation domains may be reduced in this way.

3.3 Fitting Composite References.

If there's a need for reflecting any object composition alternative (presented in section I), a "customization" of types should be performed. To customize means to adequate a pure type (selected from the universe) to the requirements specification.
For the Football Association schema, this is a Football_Team type fit:

customize type Football_Team

with

football_players: independent exclusive
coach: independent heterogeneously shared

endcustom

The exclusive clause states that the football players are a component part of a Football Team, and cannot be (at the same time) a component part of any other object. The independent clause means that the football players could exist in the database, no matter the team they belong to exists or not.

The fact of the coach being heterogeneously shared means that he can be a component part of other entities (of different type from Football Team) i.e. his family, a Football Association Comitee, and others; but he cannot be another football team's trainer.

3.4 Schema Definition.

When defining the schema for the Football Association, the relevant types Person, Football Player, Footballers Set and Football team are selected from the universe. Also the Supervisor and the Goaler are included in the schema. Should be noticed that several instances of the relationship Match-date constitute the championship. Each instance of this relationship is a relation containing pairs of football teams (which represent the matches for that date).

Schema Football_Association with

Types: Person, Football_Player, Footballers_Set, Football_team, ...
relations: Match-date, ...
axioms: ...

endschema.

3.5 Multiple Implementations

The generic type Set, may have the following associated implementation hierarchy:

SET

/ \ LIST-BASED_SET TREE-BASED_SET

/ \ ARRAY-BASED POINTER-BASED
LIST-BASED_SET LIST-BASED_SET

4. CONCLUSIONS

Differing from several existing OODB models, this proposal presents the database schema building in a separate layer from the type universe construction. An additional step of first selection and later adequation and inclusion of types in the schema is provided. This distinction of layers allows reusability and abstraction. Generic or concrete existing types and relationships are highly reusable.

Schema construction and implementation layers are independent one from the other. A type implementation does not depend on the adjustments made to the type for a special application, and vice versa. Implementations present then the same degree of reusability as types belonging to the definition layer does.

REFERENCES