Formally Speaking About Schemata, Bases, Classes and Objects

Carlos A. Tau
Universidad Nacional de La Plata, and
Comisión de Investigaciones Científicas (CIC)
calle 64 nº 520. (1900) La Plata,
Buenos Aires, Argentina.
lifia@cepivm2.unlp.edu.ar

Claudia Pons
LIFIA, Departamento de Informática,
Universidad Nacional de La Plata.
cpons@ada.info.unlp.edu.ar

Clara Smith
Universidad Nacional de La Plata, and
Consejo Nacional de Investigaciones Científicas y Técnicas (CONICET)
csmith@ada.info.unlp.edu.ar

Ana Maria Monteiro
LIFIA, Universidad Nacional de La Plata, and
DCC Universidade Estadual de Campinas,
Campinas SP, Brasil
anammont@dcc.unicamp.br

Abstract

In the early 70's, Dana Scott and Christopher Strachey laid the foundations for the theory of denotational semantics of traditional programming languages. The aim pursued in this paper is to show how concepts such as semantic domains, semantic interpretation functions and denotations can be used to describe facilities for handling non-conventional info that combines object-orientation and database management. This paper also helps to understand the expressiveness of an object manipulation language (OML).

1 Introduction

By the simple addition of persistence to object-based and object-oriented languages as Copeland and Maier did [13] the notion of object-oriented databases (objectbases) was born. But the information in any objectbase need not only to be specified (in terms of object types, by means of abstraction mechanisms) and stored (in terms of object identifiers bound to values), but also manipulated (in terms of objects). By object manipulation we mean: retrieval, deletion and modification of data stored in an objectbase and insertion of new information.

Languages for objectbases are becoming a central subject in the database area, examples are [1] [19] [3] [7] [20] [6] [24] [12], to name a few. The current tendency in the database programming area is the integration of database features with fully universal programming constructs. For a great number of commercial objectbase products and several prototypes, a set of basic manipulation operations has been developed, conforming an assorted set of OMLs. Many researchers have worked on the description of commands for objectbase manipulation. For example, recently the Object Database Management Group ODMG-93 introduced a proposal for a standard Object-Oriented Database System (but Kim [18] states that it is not a standard). They give the abstract syntax of some ODL/OML and OQL facilities [11], and assume as a principle that their formal semantics are easy to define. To achieve precision, a specification must be written in a language which has a formal basis [16]. We have chosen the denotational semantics formalism [22] for giving semantics to OML operations. This formalism is a methodology for giving mathematical meaning to programming languages and systems (as it has been done with Pascal, Ada, Lisp) combining mathematical rigor and notational elegance. Thus, the core OML operations presented are given validity, and provide foundations for further research. We emphasize their descriptional rather than their implementational aspects.

During the design of an OML, formal specifications of semantic domains, and the meaning of syntactic constructs in terms of those semantic domains, help to clarify and verify the language designer's ideas and intentions. During the language implementation stage, formal definitions are not just an univocal description of the designer's intentions, but a prescription that the language programmer must follow strictly. Thus, when the language is used, its formal description will help in checking consistency. We think formal methods for specification and design help on the development of reliable software, and give a precise description of concepts independently from any implementation, offering a basis for testing correctness.
The purpose of the paper is to understand and clarify object manipulation concepts within a mathematical framework. In section 2, we survey the main aspects of SIGMA, our reference objectbase model. Section 3 introduces a formal description of SIGMA concepts using notions belonging to the denotational semantics formalism. In section 4 we present the basic notions for object handling. We improve the denotational semantics presented in [25] for some core OML operations, making them more clear and precise. Finally, our conclusions are exposed in section 5.

2 The SIGMA Object Model

This section reviews the main characteristics of the SIGMA formal model for objectbases. SIGMA has been developed at the Universidad Nacional de La Plata in Buenos Aires, Argentina.

We identify three conceptual layers in SIGMA [4], which will be our reference objectbase model: the definition layer, the schema construction layer and the implementation layer.

The definition layer consists of a type universe with algebraic specifications of object types organized in a directed acyclic graph through is-a and is-part-of relationships, and specifications of relation types among objects. The schema construction layer is bound to a specific real world situation: for a particular application; we do not assume that the type hierarchy in the definition layer represents the objectbase schema. To build an objectbase schema, we should select types and relations of interest from the definition layer and add them to the schema layer. We may also want to add specific axioms to types, relations and the schema as a whole, according to particular cases. Finally, the implementation layer gathers implementations for types in the type universe.

While the schema layer describes every object (and relationships among objects) that may exist in a given objectbase, all objects (and relations) that effectively exist in the objectbase in a given time appear in an object universe. This universe should be understood as the physical store of objects, namely the concrete base. One single SIGMA schema may serve as the conceptual reference for several bases. In a base, objects sharing some common features are categorized into classes. Every type in the schema has an implicitly associated class in the base, called basis class, which represents the set of all instances of the same type. SIGMA also supports heterogeneous classes [5], that is, collections of objects -perhaps of distinct types- satisfying a discriminant predicate.

3 Formal Description of SIGMA Concepts

We assume the reader is familiar not only with object-oriented concepts but also with notions belonging to the denotational semantics formalism [14] [15] [23]. Our objective in this section is to give the formal framework in which some core object manipulation commands will be defined.

In this approach, the syntactic entities are interpreted within certain mathematical structures called semantic domains. In denotational semantics, domains are usually understood to be complete partially ordered sets (cpos).

The semantic domain where SIGMA objects become significant is the set of records in a similar way as Cardelli proposes in [9] and [10]. More precisely, we are talking about the domain of functions mapping labels into values, objects and procedures. We consider the set of labels partitioned in three disjoint subsets: i) feat-labels (associated to descriptive characteristics or features), ii) part-labels (connecting the object parts if any) and iii) behavior-labels (bounded to procedures representing the object's behavior). Figure 1 shows the definition of semantic domains for objects, using the primitive domains NAT and BOOL [26].

- BVALUES = NAT + BOOL, primitive domain for basic values.
- LABELS = FEAT_LABELS + PART_LABELS + BEHA_LABELS, primitive domain for names of features, parts and behavior of objects.
- FEATURES = DVALUES, domain for the observable qualities of objects.
- BEHAVIOR = [OBJECTS → OBJECTS], domain for procedures modifying objects' states.
- BOBJECTS = [LABELS → (FEATURES+BEHAVIOR)], domain for atomic objects.
- PARTS = BOBJECTS + COBJECTS, domain for constituent parts of composite objects.
- COBJECTS = [LABELS → (PARTS + FEATURES + BEHAVIOR)], domain for composite objects.
- RELATIONS = COBJECTS, domain for relationships among objects.
- OBJECTS = BOBJECTS + COBJECTS + RELATIONS, domain for objects.
- DATA = BVALUES + OBJECTS, domain for data in a base.

Fig. 1 - Semantic domains for objects
An object type specification (OTS) is a generic (parameterized) definition of objects with similar characteristics. Such specification describes a pattern, i.e., a function building a new object from other existing objects. By substituting formal parameters in a pattern we obtain either an instance of the corresponding type (if the parameters satisfy the axioms defined in the type specification) or an error (if they do not satisfy them).

The semantics for an OTS is the set of all patterns that respond to such specification. (Notation: let $S_p$ be a specification, then $\text{Models}(S_p)$ is the set of patterns described by $S_p$). Figure 2 shows the definition of a semantic domain for patterns.

$$\text{PATTERNS} = \{ \text{DATA*} \to \text{OBJECTS} + \{ \text{error} \} \}$$

**Fig. 2 - Semantic domains for object type specifications**

A type universe is a set of OTSs. This universe is a function that associates type names with algebraic specification texts. The definition of a semantic domain for type universes using the primitive domain SPEC [26] looks as in figure 3.

- SORTS, primitive domain for type names
- SPEC, primitive domain for algebraic specification texts
- $\text{TYPE\_UNIVERSES} = \{ \text{SORTS} \to \text{SPEC} + \{ \text{undef} \} \}$

**Fig. 3 - Semantic domains for type universes**

We define an implementation universe $I$, connected to a type universe $T$, as a function that for each OTS in $T$ provides a finite set of well-formed implementations for that type. This brings up the concept of multiple implementations for a type, that is:

$$\forall \text{sort } s \ (T_s \neq \text{undef} \to I_s \subseteq \text{Models}(T_s)).$$

Figure 4 shows the definition of a semantic domain for implementation universes.

- $\text{IMPL\_UNIVERSES} = \{ \text{SORTS} \to \{ \text{PATTERNS} \} + \{ \text{undef} \} \}$

**Fig. 4 - Semantic domain for implementation universes**

A SIGMA schema is a tuple $\langle e, T, I, \Omega, A \rangle$; where $e$ stands for the schema name, $T$ is the universe of selected types and relationships, $I$ stands for its implementation universe, $\Omega$ is a function providing customizing axioms for OTS in $T$, and $A$ is the set of general integrity constraints over the whole schema. Figure 5 depicts the definition of a semantic domain for SIGMA schemes.

### 4 SIGMA’s Object Handling Facilities

In this section we introduce some OML commands for expressing typical data handling operations (create, delete and change, among others). We recall here that SIGMA’s object definition language (ODL) is an algebraic specification language used to define the OTSs that belong to the $\text{TYPE\_UNIVERSES}$ domain. Its main features are formally outlined in [4]. It allows extensible type and relationship definitions and it is not tied to particular implementation languages.

Differing from the ODMG-93 proposal [11], which does not includes a standard OML, we provide a core set of operations that can be easily cast into C++, SmallTalk or any implementation language. Based on the denotations we give, the construction of a prototype is straightforward. Possible implementation strategies can be derived from the equations as well.

We formalize the meaning of the following taxonomy of commands:

- **Schema handling**: creation and deletion of a schema, and basic dynamic schema updates.
- **Base handling**: creation and deletion of a base for a particular schema, and active (current working) base setting.
- **Class handling**: creation and deletion of classes (homogeneous, heterogeneous, basis or non-basis).
- **Object handling**: creation of an object (atomic or composite), deletion of an object (atomic or composite), message sending.

What follows is the syntax for some SIGMA’s OML commands. An experienced eye may probably tend to criticize it since, as Atkinson and Buneman say [2], new incorporated ideas are difficult to cast in the syntax of well-known Pascal or SQL-like statements. We agree with them that the syntax itself is not a major issue. More important to the ease of understanding is the simplicity and the semantic consistency of the operations (which can be surely cast in a future standard OML).

- AXIOMS, primitive domain for the set of predicates
- CUSTOM = $\{ \text{SORTS} \to \text{AXIOMS} + \{ \text{undef} \} \}$, domain for the customizing axioms bound to specific types.
- $\text{SCH} = \{ \text{ID} \times \text{TYPE\_UNIVERSES} \times \text{IMPL\_UNIVERSES} \times \text{CUSTOM} \times \text{AXIOMS} \}$, domain for schemes.
SIGMA's OML syntax:

\[
\begin{align*}
\text{exp} & ::= \text{true} \mid \text{false} \mid 0 \mid 1 \mid \text{id} \mid \text{exp} \oplus \text{exp} \\
& \mid \text{exp.f} \in (\text{FEAT\_LABELS}, \ldots, \text{PART\_LABELS}) \\
\text{op} & ::= + \mid - \mid \ast \mid \text{and} \mid \text{or} \mid - \\
\text{com} & ::= \text{schema} \mid \text{base} \mid \text{class} \mid \text{object} \mid \text{com} \mid \text{com} \\
\text{schema} & ::= \text{DEL\_SCHEMA} \mid \text{update} \\
\text{updates} & ::= \text{EXPLICIT} \mid \text{id} \mid \text{EXTINCT} \mid \text{id} \\
& \mid \text{UNBOUND} \mid \text{id} \\
\text{blend\_com} & ::= \text{INSERT} \mid \text{id} \\
\text{base} & ::= \text{NEW\_BASE} \mid \text{FOR} \mid \text{SCHEMA} \mid \text{id} \\
& \mid \text{SET\_BASE} \mid \text{id} \mid \text{DEL\_BASE} \mid \text{id} \\
\text{class} & ::= \text{NEW\_CLASS} \mid \text{GROUPS} \mid \text{SATISFYING} \mid \text{exp} \\
& \mid \text{DEL\_CLASS} \mid \text{id} \\
\text{object} & ::= \text{creation} \mid \text{DEL} \mid \text{id} \mid \text{id} := \text{exp} \\
& \mid \text{exp.b} \mid \text{b} \in \text{BEHA\_LABELS} \\
\text{creation} & ::= \text{NEW} \mid \text{s} \mid \text{exp} \\
& \mid \text{NEW} \mid \text{s} \mid \text{exp} \mid \text{PARTS} \mid \text{exp} \\
& \mid \text{s} \in \text{SORTS} \\
\end{align*}
\]

The intuitive meaning of most of the above constructions should be clear. For the basic expressions, the strings \text{true} and \text{false} denote the Boolean values \text{tf} and \text{fJ} as \text{0} and \text{1} stand for the naturals \text{0} and \text{1}. Both complex Boolean and numeric expressions are built using the operators +, -, *, and, or. The operator \text{=} tests object equality (we adopt identical equality \[8\]). The string exp.f builds observable path expressions (disallowing method invocation in their bodies in order to provide a safe framework for queries). We will not discuss now whether they should contain multi-valued expressions inside or not.

The strings representing schema handling commands are: deletion of a whole schema (without affecting the type universe from which it was derived) and elementary dynamic schema updates \[21\]: type deletion, type insertion and explicitness of is-a relationships between existing types.

Base handling commands allow the creation of a base for a given schema (conceptually identical to the process of schema creation), deletion of a whole base and base activation.

Class handling commands permit the creation and deletion of non-basis classes (heterogeneous or not) in the active base. It is worth to be noticed that we can use observable path expressions (a safe kind of query) to build the SATISFYING predicate.

Strings \text{NEW} and \text{NEW PARTS} symbolize the operations for creating new atomic and composite objects respectively. The remaining object handling commands stand for deletion of objects (atomic and composite), assignment and method invocation.

The ; operator stands for command sequencing.

We use the semantic domains in figure 6 for giving semantics to expressions and commands. The reader should particularly notice that the \text{INSERT} command is not an ordinary \text{com}, it belongs to a separate syntactic category, which we call \text{blend\_com}. This category is used to denote commands that interact with info belonging to different layers. We also describe functions conforming the denotational semantic domains; using a variant of MacCarthy's formalism, a functional language based on the Lambda notation.

- \text{ID}, primitive domain for variables.
- \text{OIDS}, primitive domains for object identifiers.
- \text{EVALUES} = \text{OIDS} + \text{BVALUES}, domain for expressible values.
- \text{ENV} = \{ \text{ID} \to \text{EVALUES} + \{\text{unbound}\}\}, domain for environments. An environment bounds variable names to values
- \text{STORE} = \{ \text{OIDS} \to \text{OBJECTS} + \{\text{unused}\}\}, domain for stores. Intuitively, a store holds persistent objects.
- \text{CLASS} = \{ \text{EXP} \times \text{OIDS}\}, a class is an expression together with a set of OIDs. Every OID in this set establishes the identity of an object in the objectbase that makes the discriminant predicate true.
- \text{CLASS\_SYS} = \{ \text{ID} \to \text{CLASS} + \{\text{unbound}\}\}, domain for class systems. A class system is the set of all classes defined over the objects in a single base.
- \text{COMP\_ENV} = \{ \text{OIDS} \times \text{OIDS} \to \text{KIND\_COMP} + \{\text{unbound}\}\}, domain for composition environments. These environments gather current composition references. In the above expression, \text{KIND\_COMP} = \{\text{xid}, \text{xi}, \text{sd}, \text{si}\} stands or distinct characterizations of the composition relation: exclusive dependent/ independent, shared dependent/independent.
- \text{BASE} = \{ \text{ID} \times \text{COMP\_ENV} \times \text{STORE} \times \text{CLASS\_SYS}\}, domain for bases (object store with a class system and a composition environment over it).
- \text{OBASE\_SYS} = \{ \text{SCH} \times (\text{BASE})*\}, domain for objectbases. One single SIGMA objectbase is a list of SIGMA schemes, each of which governs a list of bases.

\[\text{Fig. 6 - Semantic domains for expressions and commands}\]\n
311
In addition, we define a family of semantic interpretation functions for the syntactic categories EXP, COM and BLEND_COM (figure 7). We use both expressions 
\[
<e,T,I,O,A>
\]

and \( \xi \) to denote elements of SCH; the symbol \( \delta \) denotes elements belonging to ENV; \( <i,\pi,\sigma,\zeta> \) and \( \mu \) indicate elements of BASE; and \( \beta \) refers to elements of OBASE-SYS.

### 4.1 Semantics for Expressions (EXP)

We give an interpretation for observable expressions. The semantics for the other exps can be deduced easily.

\[
E[e,f]_\delta = \lambda \xi, \lambda \mu. \text{Cases (} E[e]_\delta \xi \mu) \text{ of }
\]

\[
\langle \nu, \langle i, \pi, \sigma, \zeta > \rangle : \text{Cases is_oid}(\nu) \text{ of }
\]

\[
\text{tt: Cases } \sigma_y \text{ of }
\]

\[
\text{unused: error }
\]

\[
\alpha: \text{Cases } type_of(\alpha) \text{ of }
\]

\[
\text{feat_label: } <\alpha(f), \langle i, \pi, \sigma, \zeta > \rangle
\]

\[
\text{part_label: } \langle \text{oid}(\alpha(f)), \langle i, \pi, \sigma, \zeta > \rangle
\]

\[
\text{otherwise: error }
\]

\[
\text{ff: error }
\]

error: error

The string exp.f builds expressions such as Obs\( \nu \) (Obs\( \nu \) (Obs\( \nu \) (sel) ...), \( n \geq 0 \)); where sel stands for a selector [17] (which can be either an object or a variable ranging over objects), and Obs\( \nu \) ... Obs\( \nu \) are observable operations (either feature or part labels). Expression \( e \) must indicate an OID \( \nu \) both in the environment and in the base. The object (record) \( o \) is bound to \( \nu \) in store \( \sigma \). \( \sigma(f) \) is the value bound to label \( f \) in record \( o \). This value is either a BVALUE (if \( f \) is a feat_label) or an object (if \( f \) is a part_label); in this case \( e \) returns the object's OID.

### 4.2 Semantics for Commands (COM, BLEND_COM)

Operations denoted in this section are either constructive or destructive. This classification depends on whether they create or delete (any kind of) objectbase info. Particularly, we give the denotation of two commands that cannot be categorized as being fully destructive: UNBOUND and DEL-CLASS. Both commands, although destroy (in different manners) existing classes, do not delete any object in the bases.

Due to space limitations, we leave the semantics of SET_BASE and exp.b undefined. The former because of its simplicity. The latter, although its denotation is relatively easy to express, its explanation and justification exceeds the scope of this paper.

#### 4.2.1 Schema Commands

At this point, we should recall that schemas are defined (created) using SIGMA's ODL. The schema-builder command, namely SCHEMA, is used when we work on the schema construction layer for an objectbase (see section 2, or [4] for formal semantics). The following command deletes a whole schema layer:

\[
C[\text{DEL_SCHEMA } s]_\delta = \lambda \beta. \text{Cases assoc_bases}(\beta, s) \text{ of }
\]

\[
\text{lt: Cases is_empty}(\mu) \text{ of }
\]

\[
\text{ff: error }
\]

error: error

Function assoc_bases returns the (list of) \( s \)-governed bases (\( s \) should not be base-governor at its deletion time). The desired schema is then filtered from the objectbase \( \beta \).

Given two schema types \( c \) and \( s \), we can establish an is-a link between them if the \( c \)'s OTS subsumes \( s \)'s OTS. If this happens, \( c \)'s basis class is extended with objects belonging to \( s \)'s basis class.

\[
C[\text{EXPLICIT } c, s]_\delta = \lambda \beta. \text{Cases active_schema}(\beta) \text{ of }
\]

\[
\xi: \text{Cases subsumes}(\xi, c, s) \text{ of }
\]

\[
\text{error: error }
\]

error: error

where:

\[
l_\mu = \text{assoc_bases}(\beta, \text{name_of}(\xi))
\]

\[
P s <\xi, l> = (\text{name_of}(\xi) = s)
\]

Function assoc_bases returns the (list of) \( s \)-governed bases (\( s \) should not be base-governor at its deletion time).
Definition 4.2.1: The \( \leq_R \) partial ordering (read less or equally defined than [21], strongly inspired in [2] and [9]) over the set of records is defined as follows:

\[
\begin{align*}
r \leq_R r' & \text{ iif:} \\
\text{• for every feature } l, & \text{ } l \text{ is undefined in } r \text{ (has no value bounded) or, if defined, has the same value as the corresponding feature in } r'. \\
\text{• for every component object } l, & \text{ (part), such component is less or equally defined than the corresponding part in } r' \text{ (recursively).} \\
\text{• for every behavior operation } l, & \text{ the function } l \text{ is less or equally defined than the corresponding function in } r'. \text{ That is: if } is_a(\sigma, \sigma') \text{ and } is_a(\tau', \tau) \text{ then } \sigma' \rightarrow \tau' \leq_R \sigma \rightarrow \tau, \text{ and read less or equally defined than.}
\end{align*}
\]

Consequently, if \( s \leq_R c \) we consider that, restricted to the set \( L \) of labels in \( c \), objects belonging to \( s' \) basis class also belong to \( c' \)s basis class. That is:

\[
\forall o \in s, \exists o' \in c \mid labels(o) \leq_R 0'.
\]

For every \( name_of(\xi) \)-governed base the (old) class system \( \xi \) is replaced with a new one, in which \( c' \)s basis class boundary is expanded to capture \( s' \)s basis class objects.

Commands for inserting and deleting types in a given schema obviously produce a significant impact on the contents of the governed bases. SIGMA's OML includes operations for creating (inserting), deleting and updating data collected in the bases, in response to schema updates, once the governed bases are populated.

Type deletion can be stimulated by two factors: i) the conceptual extinction of objects of a given type; or ii) the need of a supression of a level of abstraction (that is, the point of view used to denote some objects).

The extinction process consists in the deletion of a schema type that, not only has no current instances in the base, but also will have no more instances from now on (they were extinguished). Type extinction is not an automatic deletion, but a schema surgery decision. The extinction process is fully described in [21].

\[
C[\text{EXPLICIT } n]_S = \lambda \beta \cdot \text{Cases active_schema}(\beta) \text{ of } \xi; \langle \xi, \beta[\text{ ext}(\xi, n) \rightarrow \xi, \text{ map } E \text{ n } \mu \rightarrow \mu] > \\
\text{error: error}
\]

where:

\[
\begin{align*}
\mu &= \text{assoc.bases}(\beta, name_of(\xi)) \\
E n &= BASE \rightarrow BASE \\
E n \langle i, \pi, \sigma, \xi \rangle &= \langle i, \pi, \pi, \sigma, \tau \rangle, \text{ and } P n \langle c, \langle s, e \rangle \rangle &= \text{type_of}(c) \leq_R \text{type_of}(n)
\end{align*}
\]

Function \( \text{ext} \) drops type \( n \), together with its subtypes, from the active schema \( \xi \). For every \( name_of(\xi) \)-
governed base, the (old) class system \( \xi \) is replaced with a new one which differs from its antecessor in the fact that class \( n \) has been filtered. Class \( n \) should not be object-container at its extinction time. The extinct operation does not alter any \( \sigma \) stores.

The unbounding process consists in the deletion of an OTS in the schema when a level of abstraction needs to be supressed. Our goal is to free instances from the class boundaries. Instances of the unbounded type not only remain in the base, but also still satisfy their original specification (they preserve their features, parts and behavior although some of them are unobservable). If an unbounded type is later reinserted in the schema, its original objects will be gathered together again.

\[
C[\text{UNBOUND } n]_S = \lambda \beta \cdot \text{Cases active_schema}(\beta) \text{ of } \xi; \langle \xi, \beta[\text{ ext}(\xi, n) \rightarrow \xi, \text{ map } U n \mu \rightarrow \mu] > \\
\text{error: error}
\]

where:

\[
\begin{align*}
\mu &= \text{assoc.bases}(\beta, name_of(\xi)) \\
U n &= BASE \rightarrow BASE \\
U n \langle i, \pi, \sigma, \xi \rangle &= \langle i, \pi, \sigma, \xi[\text{unbound/n}] \rangle
\end{align*}
\]

The unbound operation removes the \( n \)-categorized objects' boundaries provided the unbounded type is automatically replaced either i) by a supertype (if there are only objects of the unbounded type in every place objects of the deleted type are expected) or ii) a subtype (if every operation of the deleted type used by client types were those the unbounded type inherited from its substitute). This is done only when OTS subsumption can be proved. Functions \( \text{repl_super} \) and \( \text{repl_sub} \) respectively take care of this processes. Unbounding a class does not bring any change to objects; the corresponding store \( \sigma \) is not updated.

When a new type, say \( n \), is inserted in the schema, the \( B[\text{INSERT } n] \) (blend) command operates as follows:

\[
\begin{align*}
B[\text{INSERT } n] &= \lambda \beta \cdot \lambda T \cdot \lambda I. \text{Cases } T_n \text{ of } \\
\text{undef: error} \\
\text{spec: Cases } I_n \text{ of } \\
\text{undef: error} \\
\text{impls: Cases active_schema}(\beta) \text{ of } \\
\text{error: error}
\end{align*}
\]

where:

\[
\begin{align*}
\mu &= \text{assoc.bases}(\beta, name_of(\xi)) \\
E n &= BASE \rightarrow BASE \\
E n \langle i, \pi, \sigma, \xi \rangle &= \langle i, \pi, \pi, \sigma, \tau \rangle, \text{ and } P n \langle c, \langle s, e \rangle \rangle &= \text{type_of}(c) \leq_R \text{type_of}(n)
\end{align*}
\]

Function \( \text{ext} \) drops type \( n \), together with its subtypes, from the active schema \( \xi \). For every \( name_of(\xi) \)-
governed base, the (old) class system \( \xi \) is replaced with a new one which differs from its antecessor in the fact that class \( n \) has been filtered. Class \( n \) should not be object-container at its extinction time. The extinct operation does not alter any \( \sigma \) stores.

The unbounding process consists in the deletion of an OTS in the schema when a level of abstraction needs to be supressed. Our goal is to free instances from the class boundaries. Instances of the unbounded type not only remain in the base, but also still satisfy their original specification (they preserve their features, parts and behavior although some of them are unobservable). If an unbounded type is later reinserted in the schema, its original objects will be gathered together again.

\[
C[\text{UNBOUND } n]_S = \lambda \beta \cdot \text{Cases active_schema}(\beta) \text{ of } \xi; \langle \xi, \beta[\text{ ext}(\xi, n) \rightarrow \xi, \text{ map } U n \mu \rightarrow \mu] > \\
\text{error: error}
\]

where:

\[
\begin{align*}
\mu &= \text{assoc.bases}(\beta, name_of(\xi)) \\
U n &= BASE \rightarrow BASE \\
U n \langle i, \pi, \sigma, \xi \rangle &= \langle i, \pi, \sigma, \xi[\text{unbound/n}] \rangle
\end{align*}
\]

The unbound operation removes the \( n \)-categorized objects' boundaries provided the unbounded type is automatically replaced either i) by a supertype (if there are only objects of the unbounded type in every place objects of the deleted type are expected) or ii) a subtype (if every operation of the deleted type used by client types were those the unbounded type inherited from its substitute). This is done only when OTS subsumption can be proved. Functions \( \text{repl_super} \) and \( \text{repl_sub} \) respectively take care of this processes. Unbounding a class does not bring any change to objects; the corresponding store \( \sigma \) is not updated.

When a new type, say \( n \), is inserted in the schema, the \( B[\text{INSERT } n] \) (blend) command operates as follows:

\[
B[\text{INSERT } n] = \lambda \beta \cdot \lambda T. \lambda I. \text{Cases } T_n \text{ of } \\
\text{undefined} \text{ error} \\
\text{spec} \text{ Cases } I_n \text{ of } \\
\text{undefined} \text{ error} \\
\text{impls} \text{ Cases active_schema}(\beta) \text{ of } \\
\text{error: error}
\]

where:

\[
\begin{align*}
\mu &= \text{assoc.bases}(\beta, name_of(\xi)) \\
E n &= BASE \rightarrow BASE \\
E n \langle i, \pi, \sigma, \xi \rangle &= \langle i, \pi, \pi, \sigma, \tau \rangle, \text{ and } P n \langle c, \langle s, e \rangle \rangle &= \text{type_of}(c) \leq_R \text{type_of}(\mu)
\end{align*}
\]
e is an expression denoting the axioms in n's OTS.

The INSERT command takes an objectbase system \( \beta \), a library \( T \) of OTS and a library \( I \) of implementations for those OTS in \( T \). Function include puts into the active schema \( \xi \) n's OTS text (spec) together with its multiple associated implementations (impls).

Every name_of(\( \xi \))-governed base is extended in its class system with the inclusion of a basis class for \( n \), gathering every (preexisting) object satisfying \( n \). Classes in name_of(\( \xi \))-governed bases are now ready for receiving new objects of type \( n \).

4.2.2 Base Commands

One single SIGMA schema may govern multiple bases. Thus, for a given schema \( s \), we can associate as many bases as we want. To create one, we must indicate its name, and the desired governor schema.

\[ \text{\textbf{DEFINE BASE} } b \text{ FOR SCHEMA} s ]_{B} = \lambda \beta. \text{Cases get_schema}(\beta, s) of \]
\[ \xi: <b, \text{(add_base} \beta (\text{new_base} b \xi) s)> \]
\[ \text{error: error} \]

Function \( \text{new_base} : \text{ID} \rightarrow \text{SCH} \rightarrow \text{BASE} \) is defined as follows:

\[ \text{new_base} b \xi = <b, p_{0}, \sigma_{0}, \zeta_{0}> \]
where:
\[ p_{0} = \lambda (x, y). \text{unbound} \] (empty composition environment),
\[ \sigma_{0} = \lambda x. \text{unused} \] (empty store),
\[ \zeta_{0} = \lambda i d. \text{is_ots}(\xi, i d) \rightarrow \langle \text{type_of}(x) = \text{id}, \{\} \rangle, \text{unbound} \]

Function \( \text{add_base} : \text{OBASE_SYS} \rightarrow \text{BASE} \rightarrow \text{ID} \rightarrow \text{OBASE_SYS} \). This function returns a base system in which base \( \mu \) is bound to schema \( s \) in the objectbase system \( \beta \). Function is_ots(\( \xi \), \( i d \)) returns \( tt \) if \( i d \) is the name for an OTS belonging to \( \xi \).

With the exception of the active base, we can delete any base associated to the active schema.

\[ \text{\textbf{DEFINE BASE} } b ]_{B} = \lambda \beta. \text{Cases is_active}(\beta, b) of \]
\[ tt: \text{error} \]
\[ ff: \text{Cases active_schema}(\beta) of \]
\[ \xi: \text{Cases base } b \in \mu \] of
\[ tt: <\delta, \beta[\text{drop} b \mu \rightarrow \mu>] > \]
\[ ff: \text{error} \]
\[ \text{error: error} \]

where:
Function \( \text{is_active}(\beta, b) \) returns \( tt \) if a base named \( b \) is currently the active base.
\[ \mu = \text{assoc_bases} (\beta, \text{name_of}(\xi)) \]

Function \( \text{drop} : \text{ID} \rightarrow \text{(BASE)*} \rightarrow \text{(BASE)*} \) takes a base name and a list of bases. It returns the same list to which the base named \( b \) has been dropped off. Function \( \text{drop} \) is defined as follows:

\[ \text{drop} b [ 1 = 1 ] \]
\[ \text{drop} b \text{base:rest} = \text{rest}, \text{if name_of(base)} = b \]
\[ \text{drop} b \text{base:rest} = \text{base:}(\text{drop} b \text{rest}), \text{otherwise}. \]

The desired base is merely disconnected from its governor schema. Classes and objects inside a dropped base are not deleted one by one explicitly; garbage collection is to be provided.

4.2.3 Class Commands

To create a new class we indicate its name, a variable name ranging over \( \text{OIDS} \), and a predicate (for non-basis or heterogeneous classes).

\[ \text{\textbf{NEW CLASS} } n \text{ GROUPS } x \text{ SATISFYING } c]_{B} = \lambda \beta. \text{Cases active_base}(\beta) of \]
\[ <i, i, \pi, \sigma, \xi>: \text{Cases active_schema}(\beta) of \]
\[ \xi: <b, \beta[<i, \pi, \sigma, \xi[(e, S) / n]> \rightarrow <i, \pi, \sigma, \xi>] > \]
\[ \text{error: error} \]

where:
\[ S = \{ \text{oid/oid } \in \text{OIDS}; e[\delta][\text{oid/x}] \xi <i, \pi, \sigma, \xi > = <i', \mu'> \} \]

Class creation is namely the definition of an association among all objects in the active base satisfying predicate \( e \) (which should be understood as an automatically generated axiomatic expression of all features, parts and behavior defined in n's OTS for basis classes). Functions active_base and active_schema both receive an objectbase \( \beta \) and return the current schema \( <i, \pi, \sigma, \xi> \) and its current governed base \( \xi \), respectively.

We remark that the notion of state is inherent to the notion of update: the expression \( \beta[<i, \pi, \sigma, \xi][(e, S) / n]> \rightarrow <i, \pi, \sigma, \xi>] \) means that in the objectbase \( \beta \) the (old) class system \( \xi \) is updated with the inclusion of the new set \( S \) of objects satisfying \( e \), denoted by \( n \).

Dropping a non-basis class produces the removal of the boundaries of the corresponding set of objects (identical to the unbounding process, but without affecting the schema). SIGMA's OML denies the deletion of basis classes as they are considered vital type extensions (even if they are empty). The deletion of a non-basis class (either homogeneous or heterogeneous) does not produce any modification to objects inside; the corresponding store \( \sigma \) is not updated.
**C[DEL_CLASS n]_S =**
Cases active_schema(P) of
  \( \xi \): Cases basis_class(\( \xi, n \)) of
    \( \xi : \lambda \beta. \) Cases active_base(\( \xi, n \)) of
      \( <i, \pi, \sigma, \zeta> : <\delta, \beta[<i, \pi, \sigma, \zeta[\text{unbound} \div n]], \rightarrow <i, \pi, \sigma, \zeta]> > \)
    error: error
tt: error
where:
  Function basis_class returns \( \text{tt} \) if \( n \) is an OTS in schema \( \xi \).

4.2.4 Object Commands
The NEW operation creates a new object according to the schema received; and includes the object into the (current) base.

**C[NEW s e PARTS 1]_S =**
Cases active_schema(P) of
  \( \xi : \lambda \beta. \) Cases active_base(\( \xi, n \)) of
    \( <i, \pi, \sigma, \zeta> : <\delta, \beta[<i, \pi, \sigma, \zeta[\text{unbound} \div n]], \rightarrow <i, \pi, \sigma, \zeta]> > \)
    error: error
tt: error
where:
  Function basis_class returns \( \text{tt} \) if \( n \) is an OTS in schema \( \xi \).

Definitions 4.2.4a: For giving semantics to expressions handling composite objects we define

- **dep_parts w \( \pi = [c / \pi_{\text{wc}} = \text{xd} \text{ or } \pi_{\text{wc}} = \text{sd}] \)**, which returns (the list of) OIDS identifying the dependent objects forming \( w \).
- **excl_ow c \( \pi = [w / \pi_{\text{wc}} = \text{xd} \text{ or } \pi_{\text{wc}} = \text{si}] \)**, which returns (the list of) OIDS identifying all exclusive c-owners.
- **shar_ow c \( \pi = [w / \pi_{\text{wc}} = \text{sd} \text{ or } \pi_{\text{wc}} = \text{si}] \)**, which returns (the list of) OIDS identifying all shared c-owners.
- **owners c \( \pi = \text{excl_ow c } \pi + \text{shar_ow c } \pi \)**, which returns (the list of) OIDS identifying all c-owners.

The above expression indicates the creation of a composite object of type \( s \). The list \( 1 \) holds the object's component parts. Each part is an expression representing either a preexisting object, or an object being created at the same time of the owner object. Function \text{eval_parts} tests expressions in \( 1 \), and obtains from the specification \( T(s) \) its kind of composition. Thus \( l' \) is a list ranging over \((\text{OIDS} \times \text{KIND-COMP})^*\). If any component part denotes an error, \text{eval_parts} returns an error.

We get the new object \( \sigma \) by the application of the generic pattern \( \text{putt} \) over both the evaluated parameter and the component parts. \( \pi' \) is the (new) composition environment obtained by the inclusion of the new composition references; and \( \zeta' \) is the (new) class system obtained by the inclusion of \( \sigma \) into system \( \zeta \). Function \text{verif_comp} returns true if no composition constraint is violated:

\[
\text{verif_comp : ((OIDS x KIND-COMP)* x COMP_ENV) } \rightarrow \text{BOOL}
\]

\[
\text{verif_comp}((\text{oid}, k): l, \pi) =
\]

\[
\text{cases } k \text{ of}
\]

\[
x, \text{xd : (#(owners oid } \pi) = 0) \rightarrow \text{tt, ff}
\]

\[
si, \text{sd : (#(excl_ow oid } \pi) = 0) \rightarrow \text{tt, ff}
\]
and

\[ \text{verif\_comp}(l, \pi) \]

Objects, once created and categorized, can be explicitly dropped off from the objectbase where they belong. The deletion process is irreversible, it only depends on the transaction successfulness.

\[ C[\text{DEL}\ i_d]_S = \lambda \beta. \text{Cases type of}(\delta_i_d) \text{ of} \]

\[ \text{bvalue}: <\delta[\text{unbound/id}], \beta> \]

\[ \text{oid: Cases active base}(\beta) \text{ of} \]

\[ \mu: <\delta[\text{unbound/id}], \beta[\text{del\_objs}\ \mu\delta_i_d \rightarrow \mu]> \]

\[ \text{unbound: error} \]

The above expression stands for the deletion of an object in a base. If the variable id is bound to a BVVALUE, the command merely updates the environment linking id to unbound; the base \( \mu \) suffers no modifications at all. If id is bound to an object then \( \mu \) is also updated. To explain this, we need the following assistant concepts:

**Definitions 4.2.4b:** Additional definitions for the object deletion process are:

- \( \text{cascade\_del\ w\ n} = \{ \sigma / \sigma \in \text{dep\ parts\ w}_\pi \text{ and} \#(\text{owners}\ \sigma\ _\pi) = 1 \} \) returns the set of OIDs which identify objects to be deleted as a consequence of the deletion of \( w \).

- \( \text{del\_objs} :: \text{BASE} \rightarrow \text{OIDS} \rightarrow \text{BASE} \), eliminates an object from the active base. For atomic objects (BOBJECTS) the deletion just means a removal from the store. For composite objects (COBJECTS) the deletion becomes a more complicated process: there is an implicit cascade deletion of dependent component objects:

\[ \text{del\_objs}\ <l,\pi,\sigma,\zeta>_\text{oid} = \text{Cases type of}(\sigma_\text{oid}) \text{ of} \]

\[ \text{bobjects}: <\pi,\sigma[\text{unused/oid}],\text{del}\ \zeta\ \text{oid}> \]

\[ \text{coobjects}: (\lambda\ <l',\pi',\sigma',\zeta'>. <l',\pi',\sigma'[\text{unused/oid}], \text{del}\ \zeta'\ \text{oid}> ) \text{ fold}\] \( \text{del\_objs}\ <l,\pi,\sigma,\zeta>_\text{oid} \) \text{ where} \( \pi'\cdot xy = (x = \text{old}) \rightarrow \text{unbound}, \pi' x y \)

We finally introduce an interpretation for the assignment command.

\[ C[\text{id} := \text{e}]_S = \lambda \beta. \text{Cases active base}(\beta) \]

\[ \mu: \text{Cases active\_schema}(\beta) \]

\[ \xi: \text{Cases E[e]}_S \mu \text{ of} \]

\[ <\nu,\mu>, <\xi[\text{v/id}], \beta[\mu' \rightarrow \mu]> \]

error: error

error: error

error: error

The assignment process modifies the environment, bounding the identifier id to the denotation of the expression e. The expression \( [\mu' \rightarrow \mu] \) indicates that, in the SIGMA objectbase \( \beta \), the (old) base \( \mu \) is replaced by the (new) base \( \mu' \).

5 Conclusions

The database literature contains many attempts to apply denotational semantics to the specification of database concepts (e.g. papers by Neuhold, Bjorner, Ohori). Despite an undoubtable mathematical value, the approach is still not much popular in the database community. On the one hand, in the database area (particularly objectbases) the main effort is directed to the casual, irregular, arbitrary and temporal aspects of the situation being modeled. On the other hand, the denotational method tries to describe the existing world precisely. If an object-oriented data model is going to be standarized during this decade; there is a need to formalize and enhance some essencial notions before strong foundations can be laid. We think the effort to give a formal definition of the semantics of core OML concepts is valuable: the denotations given in this approach can be easily cast in any implementation language and provide a basis for a construction of a straightforward prototype. Precise descriptions are not only necessary for understanding, but are also a prerequisite for planning revolutionary changes.

Acknowledgement

We would like to thank Gabriel Baum for his help.

References


Proceedings of 8\textsuperscript{VO} Simposio Brasileiro de Bancos de Dados, Recife, Brasil, 1993.


