K. R. Dittrich, editor, Advances in Object–Oriented Database Systems, LNCS 334, pages 144–149, Berlin, 1988. Springer–Verlag

Abstract Object Types for Databases

H.-D. Ehrich
Informatik/Datenbanken, Technische Universität, Postfach 3329, D-3300 Braunschweig, FR GERMANY

A. Sernadas C. Sernadas
Departamento de Matematica, Instituto Superior Tecnico, 1096 Lisboa, PORTUGAL

Abstract - *There is a need for exploring the theoretical and methodological foundations of database design and development with the intention to achieve provably correct systems and higher levels of reusability. To this end, topics rarely addressed so far in this area must be investigated, among them full incorporation of dynamic aspects, static and dynamic integrity checking, formal methods strongly backed by a sound theory, and design-in-the-large issues like modularization and parameterization. Our approach combines the object-oriented paradigm with experiences in formal methods in software engineering, especially algebraic data type theory, logical approaches to systems specification and design, and process algebra. In analogy to algebraic data type theory, we propose mathematical models for the basic notions of object, object type and abstract object type.*

1. Introduction

What is a *database object* in the sense of object-oriented databases? How are database objects put together into *object types*? What is an *abstract* object type? What is it that makes an object or object type *complex*? How can abstract object types be *specified*? How can they be *implemented*? How can we prove *correctness*, both of specification and of implementation?

These questions - and many more - have to be answered satisfactorily in order to provide a reliable fundament for object-oriented databases. There is a need for exploring the theoretical and methodological foundations of database design and development with the intention to achieve provably correct systems and higher levels of reusability.

Object-oriented databases is a rather new field, and the development seems to be somewhat different from that of object-oriented programming. The latter began as early as 1967 with the simulation language SIMULA (DMN67), but the breakthrough is usually attributed to Smalltalk-80 (GR83). According to this development, objects are highly *dynamic* entities, displaying an active behaviour. In contrast to this, the current emphasis in object-oriented databases is more on structural aspects, especially complex object structures (Lo85, DD86). Object dynamics is largely neglected. In fact, there is some confusion about terms: a system with complex objects is not quite the same as an object-oriented system, although there are some relationships. An interesting development towards a database system modelling the behaviour, not just the structure of entities is GemStone (MSOP86, MS87).

Applying object-oriented programming ideas to databases raises a number of problems, mainly concerned with long-term persistence and large collections of objects (Be87). For instance, the problem of object *identity* becomes increasingly important (KC86). Many of these problems are not well understood, there is a definite lack of theory. To a large extent, this also holds for object-oriented programming (Am86), but matters are changing. There is, for instance, the inspiring work of Goguen and Meseguer (GM87) on unifying functional, relational and object-oriented programming on the basis of logic and algebra. Specific database issues are only marginally addressed there, but the work is relevant for databases, too. An algebraic approach to object identity is given in (Eh86, EDG86, SSE87).

We are working on an object-oriented approach to the design of information systems, incorporating object dynamics, static and dynamic integrity checking, formal methods strongly backed by a sound theory, and design-in-the-large issues like modularization and parameterization (SSE87, SFSE88).

In SSE87, we propose a formal approach to the specification of *societies of interacting objects*. The structure and behaviour of each object is defined using a primitive language that also provides the means for specifying the interactions between objects through *event sharing*. An algebraic semantics for this language is outlined. As a byproduct, the Kripke interpretation structure for the invisaged logic of object behaviour is established. The specifications are organized in two layers: (a) the *universe* of objects, their attributes and data; (b) the space of the global *trajectories* and *traces* of the society of objects. *Constraints* of several kinds can be imposed at both layers. The main issue in the construction of the universe is the *naming* of all possible objects. With respect to (b), the emphasis is on the definition of the joint behaviour of objects in terms of the allowed sequences of events that may happen in their lives.

In SFSE88, we discuss the notion of an *abstract* object type, incorporating in a compact and precise form all aspects of object structure and dynamic behaviour, including concurrency. Fundamentally, objects deal with *states* and *processes*. Concepts, tools and techniques are provided for the abstract definition of objects. Each object is described as a temporal entity that *evolves* because of the events that happen during its life. Both lifeness and safety requirements can be stated and verified.

Our current endeavour is to develop an object-oriented specification language for information systems named OBLOG, combining the object-oriented paradigm with experiences in formal methods in software engineering, especially algebraic data type theory, logical approaches to systems specification and design, and process algebra. We take care to provide a mathematical semantics as well as proof methods consistent with the semantics.

For making progress in object-oriented information systems design, we feel it is essential to base specification languages and methodologies and implemetation methods on a careful study of logic and semantic fundamentals. In this paper, we concentrate on the latter, giving a brief outline of what we think is an adequate and useful model for objects, object types, and abstract object types.

2. Objects

What is an object? A common view seems to be that it is something like a *software module* with an interface of named *operations* and a hidden local *state*, employing principles of data abstraction and encapsulation. But this is not the whole story: objects are organized into *object types* which display a sophisticated *subtyping* structure, together with an appropriate *inheritance* mechanism (CW85). Moreover, there is the whole world of object dynamics: objects may be *created*, *changed* and *destroyed*, and they may have an *internal activity* of their own. And there is some mechanism of *communication* or *interaction* between objects, e.g. by means of *messages* or *event sharing*.

Taking a more abstract view, the principle properties of an object are that it has a *state* which can change as a reaction to certain *events*, and which can be *observed*, for instance by means of *attributes* with varying *values*, displaying an observable *behaviour* in time. There have been two approaches to model this in mathematical terms: (1) objects are *state machine* (GM87) or (2) objects are *processes* (HN87) in the sense of process algebra (Ho85).

Let X be a given set of *events*. Events arise from calling methods (in the sense of object-oriented programming) with actual parameters. Thus, in theory, there might be infinitely many events. Let Y be a set of *observations*. Observations arise from values of attributes: if $A=\{a_1,...,a_r\}$ is a finite set of attributes, each with an associated type $type(a_i)$, $1 \le i \le r$, then an *observation* is a set $y \subseteq \{(a_1:d_1), \ldots, (a_r:d_r)\}$ of attribute-value pairs where $d_i \in type(a_i)$ for $1 \le i \le r$ (undefined attributes do not appear in an observation). Thus, in theory, there might be infinitely many observations. By $obs(A)$ we denote the set of all observations over attributes A.

A *state machine* $M=(S,X,Y,\delta,\beta,s_0)$ consists of (possibly infinite) sets S of states, X of inputs, and Y of outputs, together with a state transition mapping $\delta:S \times X \longrightarrow S$, an output mapping $\beta:S \longrightarrow Y$, and an initial state $s_0 \in S$.

A state machine models an object with events X and observations $Y \subseteq obs(A)$. We assume δ to be partial and β to be total: not every event may happen in every state, but there exists always an observation (which might, of course, be empty). The initial state s_0 corresponds to non-existence, all other states are states of existence. So the first event applied to an object should *create* it. The further events applied to the object *update* it in some way while it exists, and the last event, if there is one, *destroys* the object by bringing it out of existence. This way, an object processes *streams* $X^{\sigma}=X^{*} \cup X^{\omega}$, i.e. finite or infinite sequences, of events. A *life cycle* of an object is a stream of events that, when starting the machine in s_0, never encounters an undefined transition and ends in s_0 or goes on forever. Note that finite life cycles can be catenated, giving rise to several incarnations of the same object. The observations along life cycles, however, do not depend on previous incarnations.

A *process* over an alphabet X is a set $\Lambda \subseteq X^{\sigma}$ of streams over X. Thus, the set of life cycles of an object is a process. Our process model for objects is based on life cycles and observations along life cycles, abstracting from internal states. To be precise, an *object* $ob=(X,A,\Lambda,\alpha)$ consists of a set X of events, a finite set A of attributes, a set $\Lambda \subseteq X^{\sigma}$ of life cycles, and a total observation mapping $\alpha:X^{*} \longrightarrow obs(A)$ saying which attributes have which values after a finite stream (in particular after a finite initial segment of a life cycle).

Given an object as a state machine, it is obvious how to derive the process model for the same object. Not quite so obvious is perhaps that it also goes the other way round (provided we make a few natural assumptions): given an object $ob=(X,A,\Lambda,\alpha)$ as an observed process, there is a canonical state machine M "implementing" it in the sense that the life cycles and observations of the state machine are exactly those of the observed process. M can be constructed using well-known automata-theoretic techniques.

The process model is more abstract than the machine model, and it does not lose essential information. By not dealing explicitly with states, it is mathematically much simpler. So we adopt it as a semantic basis for our work.

An immediate advantage is that it is easy to express the fundamental and important *is* relationship between (single) objects. For example, a patient *is* a person at the same time, showing all attributes that he/she has as a person and being subject to all events that can happen to him/her as a person. That is, patients *inherit* attributes and events from persons, and patients can have additional attributes and events, for instance those having to do with surgery. In the process model, if $ob_i=(X_i,A_i,\Lambda_i,\alpha_i)$, $i=1,2$, are objects, then ob_2 *is* ob_1 iff $X_1 \subseteq X_2$ and $A_1 \subseteq A_2$, and the following compatibility conditions between life cycles and observations hold: (1) $\Lambda_2 \downarrow X_1 \subseteq \Lambda_1$ and (2) $\alpha_1(\tau \downarrow X_1) = \alpha_2(\tau) \downarrow A_1$ for all $\tau \in X_2^*$. Here, $\downarrow X_1$ corresponds to the hiding (concealment) operator on processes and the restriction operator on traces (Ho85), and $\downarrow A_1$ restricts the observation to the attributes in A_1.

This is just a small example of the simplicity and mathematical elegance the process view provides. We only mention that, among others, also the composition of objects to complex objects as well as object interaction, e.g. by event sharing, can be studied very satisfactorily in this setting.

3. Object Types

So far, we have dealt with *single* objects. An object *type* is a set of objects which "belong together", but in which sense ? An obvious idea might be to type objects by similarity of attribute and event structure, but this is not general enough: when it comes to generalization, we want to put objects with quite different attribute and event structures into one type. So what else ? Our answer is *object identity* and *object instantiation*, i.e. a type is determined by a coherent identification system and by saying which object instance is associated with each identifier.

The importance of object identity is strongly emphasized in KC86. Identity is that property of an object that distinguishes it from all other objects. An identification system should be able to distinguish objects regardless of their content, location or addressability, and it should make it possible to share objects. KC86 advocates identification independent of attribute values and addressability. Identification by attribute values is common in the database field, and identification by addressability is common in the programming language field. There are convincing arguments that both approaches compromise identity. Rather, a surrogate-based identification concept is suggested.

Our approach is to provide identity by means of (abstract and semantically rich) *surrogates* and *naming operations*: an *identification system* or *object universe* U=(U,OP) for an object type consists of a set U of object surrogates for the objects of that type, and a set OP of naming operations by which the surrogates can be named uniquely. A simple example is to take the natural numbers as surrogates and operations like 0, successor, addition, subtraction, etc. as naming operations: each term denotes a natural number. Please note that "aliasing", i.e. different terms denoting the same element, presents no conceptual problem.

If we want semantically meaningful identification systems, they are not always such simple. For complex object types, they can be very sophisticated, reflecting the way the type is composed from other types, maybe even recursively. Very generally speaking, an identification system is a set equipped with operations, and that is exactly what a *data* type is in the sense of algebraic data type theory. This viewpoint is also taken in Eh86, EDG86, showing how database-like identification by keys can be handled in this framework. We adopt the viewpoint that *an identification system is an algebraic data type.*

An *object type* OT=$(\mathbf{U},\omega)$ consists of an object universe **U** and an object instantiation mapping ω, associating an object with each object surrogate in the carrier U of the universe.

Complex object types can be treated conveniently in this model: we can exploit the algebraic machinery of data types for building complex universes, using any parameterized data type like set, list, product, coproduct, etc. We only have to define what happens to instantiation, i.e. how the object instances are put together. For generalization, e.g., this is quite simple: the universe is defined by disjoint union of the surrogate sets with the naming operations provided by algebraic coproducts, and each surrogate keeps the object instance it had before. For aggregation, also incorporating interaction between the components of an aggregation, e.g. by event sharing, things are not much more difficult, but we cannot go into details here.

4. Abstract Object Types

An *abstract* object type is an object type "up to unessential details" like specific choice of names, etc. Mathematically speaking, an abstract object type is a *class* of object types, for instance an isomorphism class or an equivalence class with respect to some reasonable equivalence relation, expressing intuitively that any member of the class is acceptable as a concrete implementation of that type. Like in algebraic data type theory, abstract object types come about as model classes of *specifications* based on some logical calculus. Speaking about abstract object types for databases, thus, means to speak about logic-based database specification with a precise mathematical semantics in terms of model classes.

The problem with object-oriented database specification is that there is no obvious logical calculus with a well-known model theory covering all aspects that have to be specified: data types for providing value domains for attributes as well as object types with their attributes, events and processes. For data types, there is a well established theory of equational specification. For attribute structures, general first-order predicate logic can be employed with its reasonably well understood model theory. Aspects of dynamic behaviour in time can be specified in several ways: axiomatically by using (some variant of) temporal logic or constructively by using process algebra. It is not clear yet how these - or other - approaches, though well-known and successfully applied in isolated areas, can be put together to achieve the goals described here.

5. Concluding Remarks

Object-oriented database specification using formal methods presents many more problems than we can outline in this position paper. Among the semantic issues only very briefly mentioned here are all aspects of *interaction* between objects. We favour *event sharing* as the only means of interaction, specifying which event is "the same" as which other event, modelling synchronous and symmetric communication between objects. On these grounds, the notion of an *object society* as a set of interacting objects can be defined and investigated. The mathematics of objects and object types outlined above allows to treat these problems in a precise and elegant way. We are in a position to confirm experiences made elsewhere with formal methods in software engineering: striving for semantic clarity has a beneficial effect on developing languages and methods.

References

Am86 America,P.: Object-Oriented Programming: A Theoretician's Introduction. EATCS Bulletin 29 (1986), 69-84

Be87 Beech,D.: Groundwork for an Object Database Model. In SW87, 317-354

CW85 Cardelli,L.;Wegner,P.: On Understanding Types, Data Abstraction, and Polymorphism. ACM Computing Surveys 17 (1985), 471-522

DD86 Dayal,U.;Dittrich,K.(eds): Proc. Int. Workshop on Object-Oriented Database Systems. IEEE Computer Society, Los Angeles 1986

DMN67 Dahl,O.-J.;Myhrhaug,B.;Nygaard,K.: SIMULA 67, Common Base Language, Norwegian Computing Center, Oslo 1967

Eh86 Ehrich,H.-D.: Key Extensions of Abstract Data Types, Final Algebras, and Database Semantics. Proc. Workshop on Category Theory and Computer Programming (D. Pitt et al, eds.), LNCS 240, Springer-Verlag, Berlin 1986, 412-433

EDG86 Ehrich,H.-D.;Drosten,K.;Gogolla,M.: Towards an Algebraic Semantics for Database Specification. Proc. IFIP WG2.6 Working Conf. DS-2, Albufeira 1986 (final proceedings to be published by North-Holland)

GM87 Goguen,J.A.;Meseguer,J.: Unifying Functional, Object-Oriented and Relational Programming with Logical Semantics. In SW87, 417-477

GR83 Goldberg,A.;Robson,D.: Smalltalk 80: The Language and its Implementation. Addison-Wesley, Reading, Mass. 1983

HN87 Hailpern,B.;Nguyen,V.: A Model for Object-Based Inheritence. In SW87, 147-164

Ho85 Hoare,C.A.R.: Communicating Sequential Processes. Prentice-Hall, Englewood Cliffs 1985

KC86 Khoshafian,S.N.;Copeland,G.P.: Object Identity. Proc. OOPSLA'86, ACM SIGPLAN Notices 21:11 (1986), 406-416

Lo85 Lochovski,F.(ed.): Special Issue on Object-Oriented Systems. IEEE Database Engineering 8:4 (1985)

MS87 Maier,D.;Stein,J.: Development and Implementation of an Object-Oriented DBMS. In SW87, 355-392

MSOP86 Maier,D.;Stein,J.;Otis,A.;Purdy,A.: Development of an Object-Oriented DBMS. Proc. OOPSLA'86, ACM SIGPLAN Notices 21:11 (1986), 472-482

Pa72 Parnas,D.L.: A Technique for Software Module Specification with Examples. Communications of the ACM 15 (1972),330-336

SFSE88 Sernadas,A.;Fiadeiro,J.;Sernadas,C.;Ehrich,H,-D.: Abstract Object Types: A Temporal Perspective. Proc. Colloquium on Temporal Logic and Specification, A. Pnueli et al (eds.), Springer-Verlag (to be published)

SSE87 Sernadas,A.;Sernadas,C.;Ehrich,H.-D.: Object-Oriented Specification of Databases: An Algebraic Approach. Proc. 13th VLDB, P.M.Stocker, W.Kent (eds.), Morgan-Kaufmann Publ. Inc., Los Altos 1987, 107-116

SW87 Shriver,B.;Wegner,P.(eds.): Research Directions in Object-Oriented Programming. The MIT Press, Cambridge, Mass. 1987