OBJECT-ORIENTED SPECIFICATION OF DATABASES: AN ALGEBRAIC APPROACH

Amílcar Sernadas Cristina Sernadas
Department of Mathematics, IST, Av. Rovisco Pais, 1096 Lisboa Codex, Portugal

Hans-Dieter Ehrich
Institut fur Informatik, TUB, PF 3329, D-3300 Braunschweig, FRG

ABSTRACT
The importance of abstract object types (AOTs) in the field of conceptual modeling and database design is discussed. A formal approach to the specification of societies of interacting objects is proposed. The structure and behavior of each object is defined using a primitive language that also provides the means for specifying the interactions between objects through event sharing. The algebraic semantics of this language is outlined. As a by-product, the Kripke interpretation structure for the envisaged logic of object behavior 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 behavior of the objects in terms of the allowed sequences of events that may happen in their lives.

1 - INTRODUCTION

In the area of database specification there has been a constant interest in the abstract data type concepts and tools. Those concepts and tools provide the necessary means for abstraction and modularization. Moreover, they have been used for defining formally the conceptual modeling and knowledge representation approaches [SeSe85b,SeSe86]. However, as such they are not useful when dealing with **objects** as opposed to **data**. Herein, we assume that *an object is a time evolving entity* (compare this to *inert* data). Actually, databases should be called *object-bases*, since they are composed of time evolving units. Data types are useful for describing the attribute domains, but object types are more important since they are needed to describe the entities themselves, including their behavior. The interest in objects is growing [DD86], in part due to recent concerns with: the behavioral descriptions of the universe of discourse (UoD), in the area of conceptual modeling [SeSe85a,FiSe86,Che86]; the dynamic aspects of databases [MBW80,CCF82,LEG85,Lip86]; and the manipulation of rather complex entities in engineering databases [SRG83,BaBu84,DaSm86]. Indeed, a complete description of the UoD should include the definition of both the static structure and the dynamic properties of the relevant entities. On the other hand, the full definition of the database units (records and transactions) should also cover these two aspects.

The main goal of this paper is the development of the basic concepts and tools for the specification of **abstract object types** (AOTs), as a more adequate alternative to the traditional abstract data type (ADT) techniques for conceptual modeling and full database specification. The AOT approach offers several advantages over the ADT solution. Besides allowing the view of the UoD as a society of interacting agents (objects), it also provides the means for the unified treatment of records and transactions.

A very primitive language for AOT specification is proposed. Its usefulness in the conceptual modeling of entities and their interaction is illustrated. Moreover, an outline is given of its algebraic semantics. The AOT approach to database specification starts with a complete UoD description (i.e. a full description of the society of interacting objects). This description is made in two successive steps for each object class: first its structure is defined; then, its behavior is established. Although no effort has been made so far towards providing facilities for modular descriptions, the primitive language as it stands now allows local definitions of each object class that are rather independent of each other. The extensive work in Artificial Inteligence and Software Engineering (for a survey see [StBo84]) on the object definition problem was duly taken into account. However, our approach brings in several novelties that have been missing in the previous proposals. For instance, the problem of **object naming** was solved following [EDG86,Ehr86], and really **abstract descriptions** were achieved by adapting the algebraic tools already introduced for ADTs. Moreover, the interaction mechanism between objects was chosen to be as simple as possible (**event sharing**), along the lines recently advocated in the area of concurrent programming methodology.

We should stress that the presentation of the approach, the primitive working language and its algebraic semantics leaves no room for discussing other very interesting developments. Namely, the methodologial details are left out. Moreover, no attempt is made to discuss: the modular (parameterized) specification of objects and their societies; the problem of incomplete information about the objects and their situations; the aggregation of objects into complex objects; and their classification within inheritance networks. Finally, the resulting logic of societies, objects and events is only sketchily examined.

The paper is organized as follows. Section 2 contains a rather informal introduction to the proposed tools for AOT specification within the context of a very simple UoD example. Section 3 presents the algebraic semantics of the structural descriptions of the objects (naming mechanisms, state attributes and constraints). Section 4 is dedicated to the rest of the algebraic semantics (dealing with events, trajectories, traces and interactions). The complete example discussed in Section 2 is included in the Appendix.

2 - MOTIVATION

According to the object-oriented approach, in order to capture knowledge about the real world entities and their behavior, one should start by perceiving the relevant types of objects in the **universe of discourse** (UoD). For each object type it is then necessary to understand both its static and dynamic aspects. That is to say, one should describe the object's attributes and also the events that may occur during its life. As an illustration, consider a very simplified *trader's world* . In this UoD it seems worthwhile to consider at least the following **types of objects**: *CLIENT, STOCK* and *ORDER*. Note that two rather different kinds of object types were considered: **persistent**, like *CLIENT*, and **transient**, like *ORDER*. When it comes to the implementation, it is usual to implement a persistent object over a record in the database (plus associated procedures). On the other hand, transient objects tend to be implemented as transactions. At this perception stage, the classification is irrelevant since both persistent and transient object types are to be described using the same abstractions. However, it is important to notice that both kinds should be included in the UoD and that the description of transient entities as events does not seem to be acceptable.

When describing a **society** of interacting objects, it is essential to find the basic object identification mechanisms, that is to say, for each type of objects, the means for naming the different occurrences of the type. For instance, let us assume that in the *trader's world* the occurrences of *CLIENT* are identified using a **key mechanism** that maps names (strings) into clients. One might also safely assume that, similarly, there is a key mechanism that maps stock identifiers into stocks. Naturally, a more realistic description should include the object type *STOCK* as a relationship between the object types *PRODUCT* and *DEPOT*. Herein, for the sake of simplicity, it is assumed that only stocks are of interest. Note that in the simplified *trader's world* both *CLIENT* and *STOCK* have their occurrences named independently of the other objects in the society. On the contrary, orders are identified for each client by number. That is to say, when identifying an order one has first to identify the client that issued it. Hence, the identification of orders does depend on the identification of other objects in the society.

Following [SeSe85a], the object types *CLIENT* and *STOCK* are said to be **independent** and the object type *ORDER* is said to be a **characteristic** (of *CLIENT*). Characteristic object types are said to be **dependent**. Other dependent object types are also useful (such as relationships, particularizations, generalizations and aggregations), but they are not used in the simplified *trader's world.* A detailed description of the object types in the *trader's world* is presented in the **Appendix**. A preliminary version of the **OBLOG** (OBject LOGic) language is used. In OBLOG, the **UoD description** is divided into the following parts: the **structure specification** STRUCT (including the data types plus the key mechanisms and the state attributes of all object types) and the **behavior specification** BEHAV (including the events and traces of all object types).

Returning to the example, the key mechanism of the object type *CLIENT* includes the following **key map** *cli: string → CLIENT,* as well as the **key attribute** *name: CLIENT → string.* The **key population constraints** *name(cli(S))=S* for every *S* in *string* such that *cli(S)* is in *CLIENT* and *cli(name(C))=C* for every *C* in *CLIENT* just state the obvious relations between the key attribute *name* and the key map *cli.* Such relations are typical of every independent object type. On the other hand, the other key constraint (a **key individual constraint**) *upper(first(name))=true* is specific of this object type: it states that the first character of every client's name must be an upper case letter. Note that, for the sake of simplicity, the argument of the attribute is omitted in individual formulae.

Key maps and attributes do not vary in time. That is to say, they are independent of the object's situation. State-dependent attributes are introduced in the **state** part of the object type definition. The object type *CLIENT* has only one **state attribute** *count: CLIENT → integer.* This attribute maps each client **situation** into an integer (intended to represent the number of orders already issued by that client at that point of its life). A **situation** is a snapshot of the objects and their attribute values. When specifying an object type like *CLIENT,* it is sometimes necessary to impose restrictions on the values of its attributes. For instance, the constraint *(zero≤count)=true* states that the (integer-valued) attribute *count* is never below *zero.* This is an example of a **local state constraint**. In contrast, **global state constraints** (to be illustrated later on) are similar but they depend on the situation of other objects. The local state constraint *(G(N≤count)=true)* ***if*** *(count=N)* states that the value of *count* never decreases. Note the use of the temporal operator **always in the future *G*** in the latter formula, for expressing that if at some situation *count* is *N,* then it will remain greater than or equal to *N* in every subsequent situation.

It is worthwhile to mention here the difference between situations and states, as proposed in [Ehr86]. Situations are snapshots of the existing objects and their attribute values. In constrast, *states are theories representing some knowledge about situations.* If that knowledge is partial or disjunctive there will be many situations satisfying that state. Monomorphic states (having only one situation as model) appear in special cases, like definitive databases with the closed world assumption. In the monomorphic cases, the admissibility of states and state transitions can be derived from the temporal specifications; techniques are also known for enforcing state constraints (see [ELG84,LEG85]).

The third step in the description of an object is the definition of its **behavior**, in terms of the **events** that may happen in its life. In the object type *CLIENT,* one might consider at least two kinds of events: births and (order) issues. Such events are generated using *birth: → E(C)* with *C* in *CLIENT* and *issue: integer STOCK integer → E(C)* with *C* in *CLIENT.* These are formally introduced as *CLIENT*-indexed families of functions. For each client *C, birth* denotes the birth event of *C* and the function *issue* maps each triple <order-number, stock, requested-quantity> into the corresponding issue event. *E(C)* denotes the set of possible events of client *C.*

The life **trajectories** of each client are non-empty sequences of these events satisfying some general restrictions: every trajectory begins with a **creation** event; after a **destruction** event no more events are allowed; and **modification** events are allowed after the creation event until the eventual destruction event. Note that, in principle, it is useful to allow **incomplete trajectories**, that is to say, trajectories not finishing with a destruction event. Otherwise, a rather strong liveness requirement is imposed, as discussed in the end of this section.

For each particular trajectory, the set of possible **traces** (determining intermediate situations) is composed of all non-empty initial parts of the trajectory. In many cases, the observed behavior of the object indicates that some of the theoretically possible trajectories are not allowed. For instance, in the definition of the behavior of clients, the set of trajectories is restricted by the following **domain assertion** *[T>>issue(N,K,R)]* ***only if*** *(count* ***at*** *T)=pre(N)* stating that the issue event for *(N,K,R)* is allowed only if the order-number *N* is the value of *count,* just before that event, plus one. It is important to notice that we are using **trace formulae.** In such formulae, an explicit trace argument is given for every state attribute using the connective **at**.

The dependence of the state attributes on the local trace of the object is defined with **valuation** and **equivalence equations.** As an illustration of the former consider *(count* **at** *<birth>)=zero* and *(count* **at** *[T>>issue(N,K,R)])=suc(count* **at** *T)* stating that *count* is zero after the birth event and its value increases by one every time an issue is made, respectively. Valuation equations define the values of state attributes as functions of the traces. Equivalence equations (illustrated below) define equivalences between traces for the sole purpose of attribute evaluation.

When expressing either domain restrictions or valuation and equivalence equations, it is necessary to refer to traces. A very simple notation was adopted for dealing with traces: <a1,...,an> denotes the trace composed of the events *a1,...,an*; and [x>>a] denotes the trace obtained by suffixing event *a* to trace *x*.

The final part of an object type description is composed of the so called **interaction equations** that equate events of different objects (of the same or of different types). In order to illustrate the use of those equations, as well as of some other features not present in the *CLIENT* specification, it is worthwhile to examine in detail the definition of the other object types in the *trader's world: ORDER* and *STOCK.* Skipping the key mechanism, it should be clear that the state-dependent **attributes** *stk* and *req* of *ORDER* indicate respectively the target stock and the requested quantity. Their **local state constraints** are easily interpreted. It remains to discuss the **global state constraints,** such as ***there-is*** *cli.* As global constraints, they are not imposed on the local situations of each order but instead on the global situations of the *trader's society.* Those **global situations** may be obtained by "gluing together" individual local situations. Only global situations satisfying every global constraint are considered. For instance, the former global constraint states that every existing order points (through *cli*) to an existing client.

The **interaction equation** in the *ORDER* specification *C.issue(N,K,R)= ord(C,N).creation(K,R)* states that every issue event by some client is an order creation (and vice versa). Recall that *ord* maps each pair (client,number) into the corresponding occurrence of *ORDER.* This means that those events are shared by the two objects. This **sharing of events** (with the same vocabulary or not) is the only interaction mechanism allowed in the proposed framework. Informally, the interaction equations guide the construction of the **global trajectories,** by "gluing together" the local chains at the shared events. Each global trajectory is an interleaving of the local trajectories, respecting every interaction equation. Each global state is determined by the global trace of those events that have already occurred in every object life.

The local behavior of each order is very simple: after its creation, it may be either satisfied or rejected. Note that according to this local description of the order behavior it is not possible to state if the order is to be satisfied or rejected. That can only be established when describing the behavior of the object type *STOCK.* Indeed, every rejection and every satisfaction is shared by two objects: an order and a stock, as indicated in the interaction axioms of the object type *STOCK.* The domain assertion *[T>>failure(O,R)]* ***only if*** *((qoh* **at** *T)<R)=true* states that the rejection of the order (the failure of the target stock) takes place only if the requested quantity is greater than the quantity on hand just before the event. The satisfaction (delivery) takes place in the other case. One of the events may happen, but not both. This is an interesting **safety property** of orders within the context of the proposed *trader's world.*

Sometimes, it is necessary to express (as a requirement axiom or as a provable theorem) another kind of assertion: a **liveness property.** Liveness properties express that some desired events must occur. As an illustration consider the following requirement: every order will sooner or later be processed. That is to say, no order that has been created will remain for ever waiting to be rejected or satisfied. This particular requirement is not a property of the described *trader's world.* Indeed, it is reasonably easy to verify that there are global lattices of events that satisfy the entire specification and where orders exist with incomplete trajectories. A suitable revision of the specification ensuring that liveness property could be made as follows, at the level of the *ORDER* **domain:**

<creation(K,R)> ***only if***
(F after*(satisfaction(K,R)))* **at** *<creation(K,R)>*
or *(F after**(rejection(K,R)))* **at** *<creation(K,R)>*

Herein, ***F*** is the **sometime in the future** temporal operator and **after** is a (local) trace predicate allowing the identification of the last event.

Finally, it remains to comment the **equivalence equations** for traces, such as those included in the description of *STOCK*. For instance, the equivalence equation *[T>>failure(O,R)]=T* states that, for the sole purpose of state attribute evaluation, the lefthand trace is equivalent to the righthand one. Indeed, a failure does not change the value of *qoh*.

3 - STRUCTURE SPECIFICATION

The purpose of the structure specification is to define the objects with their attributes and relationships, as well as to characterize which collections of objects may exist and how they are allowed to change. This specification is organized into three successive steps: STRUCT=DT+KY+ST, where DT is the specification of the underlying data types that will be used as attribute values; KY specifies the keys, i.e. KY defines an identification mechanism for the occurrences of each object sort; and ST provides the state information on each object sort, i.e. attributes and constraints.

Data type specifications have been studied extensively in the last decade. Without going into details, we assume a monomorphic data type DATA to be given as the semantics of the specification DT (e.g. the initial DT-algebra). As expected, DATA is a generated and typical interpretation of the specification DT. It consists of a carrier set of data elements (values) for each data sort and an appropriate data operation for each operation symbol. Note that DATA satisfies the specification DT. Hence, we write DATA $\models$ DT.

Key specifications as extensions of data type specifications have been introduced in [Ehr86,EDG86] using a final algebra semantics. Herein, we employ a variant of that approach using key maps (a notion introduced in [SeSe85a]) for which an initial algebra semantics can be given. Accordingly, a key extension of DT, $KY=(\Sigma_{KY},C_{KY})$ consists of a key signature $\Sigma_{KY}=(S_{KY},\Omega_{KY})$ and a set of **key constraints** $C_{KY}=PC_{KY}\cup IC_{KY}$, where the former is the set of **population key constraints** and the latter is the set of **individual key constraints**. The key signature contains the set S_{KY} of all **object sorts**, also denoted S_{OB}, plus an S_{OB}-indexed family of key mechanisms. Each **key mechanism** $\Omega_{KY}(s)$ is a triple $<s_1...s_n,kmap, kattr>$, where $s_1...s_n \in (S_{DT}\cup S_{OB})^*$ is the list of the n key argument sorts, kmap: $s_1...s_n \rightarrow s$ is the **key map**, and $kattr = <ka_1,...,ka_n>$ is the list of the n **key attributes**, such that $ka_j: s \rightarrow s_j$ for $1 \leq j \leq n$.

The semantics U of $DT+(\Sigma_{KY},PC_{KY})$ is the free extension of DATA with respect to (Σ_{KY},PC_{KY}) forgetting the key maps. By this free construction, U satisfies the population key constraints. Intuitively, U contains for each object sort all "possible" objects of that sort (in an abstract sense) that can be identified using the specified key mechanisms.

For formulating the individual key constraints we use the **key language** KL: the first-order predicate language over the signature of DT+KY. For notational and methodological convenience, in OBLOG specifications, we employ a "local dialect" KL(s) when specifying the object type s. Formulae of the form $(\forall y:s)\ \varphi(y)$ are abbreviated to φ, omitting all occurrences of the object variable y of sort s. For instance, in the specification of *ORDER*, the formula *(zero≤numb)=true* is the local abbreviation of a formula of KL: *(∀y:ORDER) ((zero≤numb(y))=true)*. The semantics of DT+KY in the presence of such individual key constraints is somewhat involved. The starting point is the algebra U built as above ignoring those individual key constraints. Then, the envisaged semantics UNIVERSE of DT+KY (taking into account the individual key constraints) should be the largest subalgebra of U satisfying the individual key constraints. In [Ehr86,EDG86] it is proved that a unique largest subalgebra exists provided that all individual key constraints are positive formulae. We shall assume that in the sequel. Note that UNIVERSE $\models$ (DT+KY), in the sense that it satisfies the key constraints and its Σ_{DT}-reduct satisfies the DT axioms.

Finally, the **state specification** $ST=(\Sigma_{ST},C_{ST})$ completes the structure specification by providing the object attributes and their constraints. The state signature $\Sigma_{ST}=(\emptyset,\Omega_{ST})$ adds no new sorts. It introduces the **state attributes** of each object sort, as follows: $\Omega_{ST} = \{\Omega_{ST}(s): s \in S_{OB}\}$ where $\Omega_{ST}(s) = \{\Omega_{ST}(s)(s'): s' \in S_{DT}\cup S_{OB}\}$. As usual, we write b: $s \rightarrow s'$ instead of $b \in \Omega_{ST}(s)(s')$. Note that state attributes can be data or object-valued. Recall also that we assumed that an **existence** boolean attribute is available for every object sort s: **there-is**$_s$: $s \rightarrow$ **bool**. The index is omitted if it is clear from the context. The **state constraints** are formulated using the **structure language** SL: a temporal extension of the first order predicate language over the structure signature. The only temporal operators we use in this paper are the "temporal quantifiers" **always in the future G** and **sometime in the future F**. As for the key language KL, we also use a "local dialect" SL(s) for each object sort s whenever convenient. Indeed, in the specification of s, we write φ as an abbreviation of ($\forall$y:s) ((**there-is**$_s$(y)=true) **implies** φ(y)). The set of **state constraints** is given as follows: $C_{ST}=LC_{ST} \cup GC_{ST}$, where the former is the set of **local state constraints** (of every object sort) and the latter is the set of **global state constraints** (also of every object sort).

Now, at last, the semantics of the structure specification STRUCT can be given as any "temporal interpretation" satisfying the state constraints. Before defining what we mean by a temporal interpretation, the concept of **situation** should be formalized as follows: a situation σ for STRUCT is a Σ_{STRUCT}-algebra (without the key maps) whose $(\Sigma_{DT}+\Sigma_{KY})$-reduct is UNIVERSE. Then, a **temporal interpretation** for STRUCT=DT+KY+ST is a triple TI=<UNIVERSE,S,T>, where the first component is the UNIVERSE of data and objects satisfying DT+KY; **S** is a class of situations for STRUCT; and **T** is a reflexive and transitive binary relation on **S**. The latter corresponds to the transition or reachability relationship between situations: $\sigma T \sigma'$ iff the situation σ' is reachable starting from situation σ, i.e. iff the transition from σ to σ' is possible.

Given a temporal interpretation TI=<UNIVERSE,S,T>, a situation σ in **S** and a variable assignment A, every term of SL can be evaluated as usual: data operations and values are interpreted according to DATA; key attributes and objects are interpreted according to UNIVERSE; state attributes are interpreted according to the situation σ. Hence, every atomic formula can easily be evaluated, since we have only the equality predicate (for each sort): $V_{TI,\sigma,A}(t=t')$ = if $(V_{TI,\sigma,A}(t) = V_{TI,\sigma,A}(t'))$ then 1 else 0. Finally, non-atomic formulae are evaluated as expected. For example: $V_{TI,\sigma,A}(F\varphi)$ = if $(V_{TI,\sigma',A}(\varphi)=1$ for every σ' such that $\sigma T \sigma')$ then 1 else 0. Note the role of the transition relation **T** in the interpretation of temporal formulae, following the Kripke approach. It is beyond the scope of this paper to discuss the properties of that relationship in the context of database specification. A temporal interpretation TI=<UNIVERSE,S,T> is said to satisfy a structure specification STRUCT, TI |= STRUCT, iff every state constraint has the value 1 in TI for every situation s in S and every variable assignment. We shall take as the semantics KRIPKE of a structure specification STRUCT the class of temporal interpretations for STRUCT that satisfy its state constraints.

4 - BEHAVIOR SPECIFICATION

The purpose of the behavior specification is to define the life trajectories of the objects in the society, in terms of their events, as well as their interactions. This specification is organized into four successive steps: BEHAV=EV+TJ+TA+TD, where EV is the specification of the events, including the interactions, TJ establishes the set of all possible (global) trajectories and their traces, TA provides the trace attributes and their equations, and, finally, TD specifies the domain of the acceptable trajectories and their traces.

The **event specification** $EV=(\Sigma_{EV},E_{EV})$ consists of an event signature $\Sigma_{EV}=(S_{EV},\Omega_{EV})$ and a set of **interaction axioms**. An event signature consists of a single set S_{EV} with the sort **e** of events, an S_{OB}-indexed family of event generator tools and a collection of event predicates. Each $\Omega_{EV}(s)$ is a triple <creation(s),modification(s),destruction(s)>, where, for instance, creation(s) = {creation(s)(w): $w \in (S_{DT} \cup S_{OB})^*$}. Each set creation(s)(w) contains the creation-event **generator symbols** for object sort s (the sort of the life or primary parameter) with list w of secondary parameter sorts. As expected, we write g: s w $\rightarrow$ **e** when $g \in$ creation(s)(w). The event predicates **cre**, **mod** and **des** are used to identify the category of each event. When writing the interaction axioms we use the **event language** EL: the equational language over the signature of DT+KY+EV, prefixing the life parameter. As an example, recall the equation: *C.issue(N,K,R) = ord(C,N).creation(K,R).*

The semantics EVENT of the specification DT+KY+EV is the free extension of UNIVERSE with respect to the event specification EV. Thus, EVENT is a $(\Sigma_{DT}+\Sigma_{KY}+\Sigma_{EV})$-algebra (without the key maps) satisfying the E_{EV} equations and containing UNIVERSE as $(\Sigma_{DT}+\Sigma_{KY})$-reduct. Thus, EVENT |= (DT+KY+EV). The algebra EVENT contains, in addition to the components of UNIVERSE, the carrier set **E** for the event sort **e**, as well as the appropriate interpretation of each event generator and predicate.

The **trajectory extension** $TJ=(\Sigma_{TJ},C_{TJ})$ is composed of the trajectory signature $\Sigma_{TJ}=(S_{TJ},\Omega_{TJ})$ and the set of universal **trajectory conditions**: the **trajectory equations** and the **trajectory constraints**. The trajectory signature includes one sort: the sort $\mathbf{e}^+$ of sequences of events or **trajectories** and the sort **r** of **traces**. It also includes the following operation symbols: the trajectory constructors <_>: **e** → $\mathbf{e}^+$ and [_>>_]: $\mathbf{e}^+$ **e** → $\mathbf{e}^+$, as well as the trace constructor **tr**: $\mathbf{e}^+$ **int** → **r** and some auxilliary operators, such as **ev**: **r int** → **e** and **lp**: $\mathbf{e}^+$ **int** → $\mathbf{e}^+$. Note that **ev**(R,N) denotes the N-th event in trace R; and **lp**(S,N) denotes the sequence of the N first events of S. The trajectory equations written in the equational language over the signature of DT+KY+EV+TJ impose the expected relationships between trajectories, traces and events. For instance: **ev(tr(<X>,one),one)**=X. On the other hand, the trajectory constraints (easily formalized in the suitable first order language over the signature DT+KY+EV+TJ) impose the restrictions described in section 2. For instance, they include the requirement that every trajectory must start with a creation event.

The semantics TRAJECTORY of the specification DT+KY+EV+TJ is the largest subalgebra, satisfying the trajectory constraints above, of the free extension of EVENT (forgetting the trajectory constructors) with respect to the indicated trajectory extension without those constraints. It contains, inter alia, the carrier set E^+ of sort $\mathbf{e}^+$ (the set of all trajectories). Fixing a trajectory τ, that is to say, fixing an element τ of E^+, TRAJECTORY(τ) is the largest subalgebra of TRAJECTORY such that the carrier set of $\mathbf{e}^+$ is the set $\{\tau\}$. Hence, its carrier set $\mathbf{R}(\tau)$ for **r** contains all initial parts of τ.

The **trace attributes specification** TA=(Σ_{TA},E_{TA}) consists of the signature Σ_{TA}=(∅,Ω_{TA}) and a collection of **trace equations**: the **trace equivalence equations** and the **valuation equations**. The trace attributes signature adds no new sorts. It introduces a collection of **trace attribute symbols**, including: $\{\Omega_{TA}(\mathbf{r},s'): s' \in (S_{DT} \cup S_{OB})\}$, where each $\Omega_{TA}(\mathbf{r},s')$ is the set of attribute symbols mapping traces into elements of sort s'. We assume that each $\Omega_{TA}(\mathbf{r},s')$ contains every state attribute symbol (see section 3) returning values of sort s'. Hence, *we use the same symbol for a state attribute and the corresponding trace attribute,* since there is no possibility of confusion. As an illustration, recall the state attribute *qoh: STOCK → integer.* The same symbol is used in the behavior specification denoting a trace attribute that maps traces into integers.

Moreover, **after** is introduced at this point as an event-dependent boolean trace attribute with no counterpart in the collection of state attributes **after: e r → bool**. When writing trace equations we use the **trace language** TL: the equational language over the signature DT+KY+TJ+TA, suffixing the trace argument of the attributes through the connective **at**. For instance, recall the following equations in the *STOCK* specification: *[T>>failure(O,R)]=T* and *(qoh **at** <open>)=zero.* The former is a trace equivalence equation that simply equates traces (only for the purpose of attribute evaluation). The other is a valuation equation that states the value of the attribute at the indicated trace. Actually, the former equation is an abbreviation of the schema stating for each attribute att of arbitrary object *K* of sort *STOCK*: *(att **at** [T>>K.failure(O,R)])=(att **at** T).* Besides the specified trace equations, we assume some **implicit trace equations** that indicate the value of the boolean existence and event-dependent attributes, and that impose the necessary **frame conditions.** As an example of the former, consider the following valuation equation: ***(there-is K at <K.open>) = true.***

Fixing a trajectory τ in TRAJECTORY, the semantics ATTRIBUTE(τ) of specification DT+KY+EV+TJ+TA is the class of extensions of TRAJECTORY(τ) with respect to the indicated trace attributes specification (including the implicit equations). Note that each element θ of ATTRIBUTE(τ) contains TRAJECTORY(τ) as $(\Sigma_{DT}+\Sigma_{KY}+\Sigma_{EV}+\Sigma_{TJ})$-reduct. Consequently, it also contains EVENT as $(\Sigma_{DT}+\Sigma_{KY}+\Sigma_{EV})$-reduct. In this sense, we can say that any such algebra satisfies the specification DT+KY+EV+TJ+TA. Thus, we write: ATTRIBUTE(τ) |= (DT+KY+EV+TJ+TA).

Finally, it remains to define precisely the semantics of the fourth part of the behavior specification: TD defines the set of acceptable trajectories restricting at the same time the possible values of the trace attributes. The **trace domain extension** TD=(∅,C_{TD}) introduces no further sorts or operations. It consists of a collection of **domain constraints** of the general form: T **only if** φ(T), where φ(T) is a formula of TL (the trace language introduced above) and T is a term of sort $\mathbf{e}^+$. Fixing a particular trajectory τ in TRAJECTORY, such a constraint is to be interpreted as stating (∃v:**int**) **lp**(τ,v)=T **implies** φ(T). Within each algebra θ of ATTRIBUTE(τ), this assertion is either true or false. We select each algebra θ of ATTRIBUTE(τ) only if it satisfies all domain constraints. When that is the case, we write $\theta \models TD$ and, thus, by introducing the set of such algebras $DOMAIN(\tau) = \{\theta \in ATTRIBUTE(\tau): \theta \models TD\}$, we can write: $DOMAIN(\tau) \models (DT+KY+EV+TJ+TA+TD)$. Note that it is possible that for some τ the class $DOMAIN(\tau)$ is empty. Every such trajectory is said to be forbidden (by the domain constraints).

Naturally, the behavior semantics as defined above was not required to satisfy the full structure specification, since the state constraints could not be taken into account. We say that **the behavior specification complies with the structure specification** when the state constraints are satisfied by every model of the behavior specification in the following sense: $(\forall \tau \in TRAJECTORY)\ (\forall \theta \in DOMAIN(\tau))\ TI_\theta \models C_{ST}$. The (linear) temporal interpretation TI_θ corresponding to each algebra θ in $DOMAIN(\tau)$ is defined as follows: $TI_\theta = \langle UNIVERSE, S_\theta, T_\theta \rangle$, where UNIVERSE is the $(\Sigma_{DT}+\Sigma_{KY})$-reduct of TRAJECTORY, $S_\theta = \{\mu_\theta(\rho): \rho \in \mathbf{R}(\tau)\}$ and $\mu_\theta(\rho)\ \mathbf{T}_\theta\ \mu_\theta(\rho')$ iff ρ is initial part of ρ'. It remains to define the mapping μ_θ from $\mathbf{R}(\tau)$ into the class of all STRUCT-situations. From the algebra θ it is easy to build for each trace (initial part of τ) the corresponding situation. Indeed, it is the Σ_{STRUCT}-algebra that maps each state attribute symbol to the value of the corresponding trace attribute symbol at that trace as given by θ.

5 - CONCLUDING REMARKS

The rigorous algebraic semantics of an effective language for the specification of **abstract object types** (AOTs) was outlined. This semantics provides the Kripke interpretation structure for the envisaged logic of object behavior. Some of the requirements for this logic were discussed, namely concerning the proof of liveness and safety properties of the object societies. Such a language was shown to be useful when describing the universe of discourse as a society of interacting objects. This approach to conceptual modeling provides the means for the complete description of the relevant entities, including their individual and interaction behavior. This description is the starting point for the design of both the database schema and the associated transactions. Indeed, both records and transactions can and should be seen as objects, although the former tend to be more persistent than the latter. Hence, it is possible to design the database and the associated applications as a society of objects of varying persistency. This view also opens up the possibility of extending the current architectures of database management systems (DBMS) towards real **object society management systems** (OSMS).

Some interesting developments were not reported herein due to space limitations. Namely, the methodological guidelines for database design according to the proposed approach (already under experimentation) and the detailed development of the algebraic semantics concerning the key mechanisms and the object universe construction. On the other hand, further work is necessary concerning the refinement of the behavior semantics in order to deal with infinite trajectories and also to establish the desired temporal structure (either linear or branching). Afterwards, it will be possible to put together the logical calculus for proving properties of the object societies. Other important lines of research are related to the problems of dealing with incomplete information about the objects and of supervising and enforcing database state constraints. Finally, we are also looking at further developments of the language in order to allow parameterized specifications, as well as inheritance and aggregation.

Although the proposed formalism for AOT specification seems rather complex compared to the traditional formalisms for ADT (abstract *data* type) specification, we believe that it will be rather difficult to simplify it further. Indeed, objects are temporal entities rather more complex than simple values. We expect that future DBMS should incorporate basic facilities for object society management along the lines of our proposal where the chosen interaction mechanism is as simple as possible (event sharing by the interacting objects).

6 - REFERENCES

[BaBu84] Batori, D. and Buchmann, A., "Molecular Objects, Abstract Data Types and Data Models - A Framework", **Proc. VLDB Singapore,** 1984.

[CCF82] Castilho, J., Casanova, M. and Furtado, A., "A Temporal Framework for Database Specification", **Proc. VLDB Mexico City,** 1982.

[Che86] Chen, P., "The Time Dimension in the Entity-Relationship Model", **Proc. IFIP World Congress 86**, North Holland, 1986.

[DaSm86] Dayal, U. and Smith, J., "Probe: A Knowledge-Oriented Database Management System", **On Knowledge Base Management**

Systems, Brodie, M. and Mylopoulos, J. (eds), Springer-Verlag, 1986.

[DD86] Dayal, U. and Dittrich, K. (eds), **Proc. Int. Workshop on Object-Oriented Database Systems**, IEEE-CS, 1986.

[EDG86] Ehrich, H.-D., Drosten, K. and Gogolla, M., "Towards an Algebraic Semantics for Database Specification", **Knowledge and Data (DS-2)**, Meersman, R. and Sernadas, A. (eds), Proc. IFIP WG 2.6 Working Conference, Albufeira, 1986, North-Holland (to appear).

[Ehr86] Ehrich, H.-D., "Key Extensions of Abstract Data Types, Final Algebras and Database Semantics", Pitt, D. et al (eds), **Proc. Workshop on Category Theory and Computer Programming**, Springer-Verlag, 1986.

[ELG84] Ehrich, H.-D., Lipeck, U. and Gogolla, M., "Specification, Semantics and Enforcement of Dynamic Database Constraints", **Proc. VLDB Singapore**, 1984.

[FiSe86] Fiadeiro, J. and Sernadas, A., "The Infolog Linear Tense Propositional Logic of Events and Transactions", **Information Systems**, 11[1], 1986.

[LEG85] Lipeck, U., Ehrich, H.-D. and Gogolla, M., "Specifying Admissibility of Dynamic Database Behaviour Using Temporal Logic", Sernadas, A., Bubenko, J. and Olivé, A., **Information Systems: Theoretical and Formal Aspects**, North Holland, 1985.

[Lip86] Lipeck, U., "Stepwise Specification of Dynamic Database Behaviour", **ACM SIGMOD Int. Conf. on Management of Data**, 1986.

[MBW80] Mylopoulos, J., Bernstein, P. and Wong, H., "A Language Facilty for Designing Interactive Database-Intensive Systems", **ACM TODS**, 5[2], 1980.

[SeSe85a] Sernadas, A. and Sernadas, C., "Capturing Knowledge about the Organisation Dynamics", Methlie, L. and Sprague, R. (eds), **Knowledge Representation for Decision Support Systems**, North-Holland, 1985.

[SeSe85b] Sernadas, A. and Sernadas, C., "Abstraction and Inference Mechanisms for Knowledge Representation", Schmidt, J. and Thanos, C. (eds), **Foundations of Knowledge Representation**, Proc. Workshop, Xania, 1985, Springer-Verlag (to appear).

[SeSe86] Sernadas, C. and Sernadas, A., "Conceptual Modeling Abstractions as Parameterized Theories in Institutions", Meersman, R. and Steel, T. (eds), **Database Semantics (DS-1)**, North-Holland, 1986.

[StBo84] Stefik, M. and Bobrow, D., "Object-Oriented Programming: Themes and Variations", **The AI Magazine**, 1984.

[SRG83] Stonebraker, J., Rubensteinn, B. and Guttman, A., "Application of Abstract Data Types and Abstract Indices to CAD Databases", **ACM SIGMOD Database Week: Eng Design Applications**, 1983.

7 - APPENDIX: trader's world

```
object type CLIENT
   key
      map   cli(string)
      attributes
         name  :  string
      constraints
         population
            name(cli(S))=S ;
            cli(name(C))=C
         individual
            upper(first(name))=true
   state
      attributes
         count  :  integer
      constraints
         local
            (zero≤count)=true ;
            (G(N≤count)=true) if (count=N)
   behavior
      events
         creation
            birth
         modification
            issue(integer,STOCK,integer)
         traces
         domain
            [T>>issue(N,K,R)] only if
               (count at T)=pre(N)
         equations
            valuation
               (count at <birth>)=zero;
               (count at [T>>issue(N,K,R)])=
                  suc(count at T)
end

object type ORDER
   key
      map   ord(CLIENT,integer)
      attributes
         numb  :  integer ;
         cli   :  CLIENT
      constraints
         population
            cli(ord(C,N))=C ;
            numb(ord(C,N))=N ;
```

```
            ord(cli(O),numb(O))=O
        individual
            (zero≤numb)=true
  state
    attributes
        stk :   STOCK;
        req   :   integer
    constraints
        local
            (zero<req)=true ;
            (G(stk=K)) if (stk=K)
        global
            there-is cli
            there-is stk
  behavior
    events
        creation
            creation(STOCK,integer)
        destruction
            satisfaction(STOCK,integer) ;
            rejection(STOCK,integer)
    traces
        domain
            [T>>satisfaction(K,R)] only if
                (stk at T)=K and (req at T)=R ;
            [T>>rejection(K,R)] only if
                (stk at T)=K and (req at T)=R
        equations
            valuation
                (stk at <creation(K,R)>)=K ;
                (req at <creation(K,R)>)=R
  interaction equations
      C.issue(N,K,R)=ord(C,N).creation(K,R)
end

object type STOCK
  key
    map   stk(string,string)
    attributes
        prdid  :   string ;
        depid  :   string
    constraints
        population
            prdid(stk(S1,S2))=S1 ;
            depid(stk(S1,S2))=S2 ;
            stk(prdid(K),depid(K))=K
  state
    attributes
        qoh    :   integer
    constraints
        local
            (zero≤qoh)=true
  behavior
    events
        creation
            open
        destruction
            close
        modification
            delivery(ORDER,integer) ;
            failure(ORDER,integer) ;
            update(integer)
    traces
        domain
            [T>>update(R)] only if
        (minus(R)≤(qoh at T))=true ;
            [T>>delivery(O,R)] only if
        (R≤(qoh at T))=true ;
            [T>>failure(O,R)] only if
        ((qoh at T)<R)=true
        equations
            equivalence
              [T>>delivery(O,R)]=[T>>update(-R)];
              [T>>failure(O,R)]=T ;
              [T>>update(R1)>>update(R2)]=
                    [T>>update(R1+R2)]
            valuation
                (qoh at <open>)=zero ;
                (qoh at <open,update(R)>)=R
  interaction equations
      O.satisfaction(K,R)=K.delivery(O,R) ;
      O.rejection(K,R)=K.failure(O,R)
end
```