Object-Oriented Databases: Analysis, Design & Construction (DS-4)
R.A. Meersman, W. Kent, S. Khosla (Editors)
Elsevier Science Publishers B.V. (North-Holland)

WHAT IS AN OBJECT, AFTER ALL?

A. SERNADAS[1] and H.-D. EHRICH[2]

(1) Computer Science Group, INESC, Apartado 10105, 1017 Lisboa Codex, PORTUGAL

(2) Abteilung Datenbanken, Tech Univ Braunschweig, Postfach 3329, 3300 Braunschweig, FRG

The envisaged notion of object is presented as corresponding to the basic, universal building block of (information) systems. A simple mathematical model for fully concurrent objects (actors) is adopted that extends a suitable model for sequential processes. An object is defined as a process possibly endowed with initiative and trace-dependent attributes. Transactional requirements are analysed within this framework as liveness requirements. Object aggregation is explained using the general notion of object morphism. The basic inheritance, overriding and reification mechanisms are also presented, as well as a suitable notion of object-type. The computational model is shown through examples to provide a sound basis for (information) systems design, including abstract conceptual modeling and layered implementation of both passive (record-like) and active (procedure-like) objects. The model establishes a suitable semantic domain for the envisaged broad spectrum specification/design language, as well as for the associated compositional calculus for correctness verification. Methodological issues are also briefly discussed, namely concerning the object-oriented design of systems to be implemented on traditional database environments.

1. INTRODUCTION

Object-oriented programming began as early as 1967 with the simulation language SIMULA [6], but its popularization is usually attributed to the advent of Smalltalk [21]. Object-orientation has been proposed as a computation paradigm by itself [25,26]. However, after all these years it still lacks an effective mathematical foundation. We might even argue that the very notion of object is yet to be agreed upon. But many of its essential aspects have been "frozen" [1,2,7,29,37,39], to the point where it seems to have become feasible to start working towards a definite, mathematical model for objects.

We have been working towards such a model [11,13,14,31,32,34], including objects, object-types, aggregation of concurrent, interacting objects and reification of objects with objects,

capitalizing in some analogies from algebraic data-type theory as well as from process theory. Indeed, according to the object-oriented view, a stack is not a data-type, but a single *object instance*. As an object, a stack has an internal *state* that can be changed by certain operations (*events*) and observed through slots (*attributes*). On the other hand, as an object, the stack displays behaviour. That is, there is a *process* corresponding to its behaviour. Thus, basically, an object is a process endowed with trace-dependent attributes.

In section 2, the concept of object is informally reviewed and related to the underlying notion of process in order to prepare the reader for a mathematical definition within the setting of a simple model of deterministic, sequential processes. In section 3, we discuss object aggregation, first via examples and later presenting a categorial definition. At that stage it is already possible to discuss the *is-part-of* and *is-a* relationships as morphisms, including inheritance issues — section 3 and 4, respectively. Method overriding is also briefly analysed in section 4. Still within the proposed categorial framework, in section 5 we motivate and define object-type. Therein we also discuss subtyping, as well as the relationship to traditional data modeling techniques. In section 6, we outline the principles of object reification, including some transactional aspects. Finally, in section 7 we stress the role of the proposed semantic domain within the context of object-oriented approaches to (information) systems development, outlining the requirements for broad spectrum specification/design languages and calculi.

We make moderate use of the theory of categories, as well as of the theory of sequential processes. The reader may find all the necessary category-theoretic notions in the first chapters of Goldblat's book [22]. And those from the theory of processes in Hennessy's book [24], although we use a simple model of processes first proposed in [4].

2. WHAT IS AN OBJECT ?

A computer system, when observed as a whole, is a symbolic *machine* that is able to perceive, manipulate, store, produce and transmit information. As such, the computer system is composed of two basic kinds of parts. On one hand, we have the *storage components* such as files, records, databases and, of course, working areas in central memory. These storage components are able to memorize lexical things like integers, names and so on, in general known as *data*. On the other hand, we have the *process components* such as running application programs, transactions, operating systems programs and so on. These process components are responsible for the activity of the computer system. They receive, manipulate and produce all sorts of data using, whenever necessary, the storage components.

In spite of their apparent diversity, we can recognise some important common features among all these parts of the computer system. Forgetting data for the moment, both the storage and the process components have a distinct temporal and spatial existence. Any of them is created and evolves throughout time (i.e. changes its state), possibly moving from one place to another, until it is finally destroyed (if ever). Any of them is able to retain data and is able to replace the data it is storing. Any of them can either be very *persistent* (with a long life) or *transient* (with a short life).

The only intrinsic difference between a so called storage component and a process component is in its *liveness*. The former is *passive* whereas the latter is *active*. That is to say, the latter has liveness requirements and initiative in the sense that it has the ability to reach desired goals by itself (e.g. termination of program execution), whereas the former waits passively for the interactions with the surrounding active components. In traditional jargon, the latter is given CPU resources, the former is not.

Thus, we should look at all those components of the computer system as examples of the same concept - the *object* - with varying degrees of liveness and persistence. It is a shame that the Von Neumann computation paradigm has lived in this respect beyond its usefulness [27]. It imposes an artificial boundary between active and passive objects, and between persistent and transient objects. Hence, the current computer technology (both hardware and software) provides different tools for defining and supporting the four classes of objects. For instance, active transient objects are defined as procedures using a so called programming language, and are supported by the process manager of the operating system; and passive persistent objects are nowadays defined using a database schema language, and supported by the database manager.

But the recent advances in object-oriented approaches promise a near future when it will be possible to work with an "object management system" providing uniform support to all classes of objects. According to this view, the world in general, and the computer systems in particular, are societies of interacting, fully concurrent objects, some of them active, some of them passive, and with varying degrees of persistence.

In conclusion, the basic building block of computer systems is the *object* (actor), presented herein as a process with initiatives and trace-dependent attributes. Clearly, it should be possible to put together objects in order to build larger objects — aggregation. The resulting joint behaviour should correspond to the parallel composition of the behaviour of the component objects. That is, aggregation should appear as the extension of parallel composition to attributes.

Processes

For the sake of simplicity we consider herein only sequential, deterministic behaviours. That is, we postulate that the behaviour of an object is a sequential, deterministic process. Clearly, it is feasible to carry over this notion of object behaviour to other more general process models, namely dealing with non-determinism.

Thus, following [4], we assume that a *process* over a given alphabet Evt of events (atomic actions) corresponds to a set **L** of traces (sequences of events):

$$pr = (Evt, L)$$

such that

$$L \subset Evt^*$$

L contains the empty trace ε ;

$$\forall\, s,r \in Evt^* \quad \forall\, A,B \subset Evt \text{ st } A \cap B = \varnothing :$$
$$((s\ perm(A)\ perm(B)\ r) \cup (s\ perm(B)\ perm(A)\ r)) \subset L \Rightarrow (s\ perm(A \cup B)\ r) \subset L$$

where *perm*(X) denotes the set of all permutations over X, and, for instance, if $W \subset Evt^*$, the expression s W r denotes the set of traces $\{s\ w\ r : w \in W\}$.

Intuitively,

$$(s\ perm(B)\ perm(A)\ r) \subset L$$

means that the events in A may occur in any order (or "concurrently") between s and r. Hence, the constraint above on the language L imposes that if A and B are two such "concurrent" sets of events that may occur contiguously in any order, then they are also "concurrent", i.e. they may occur arbitrarily interleaved.

In this context we say that Evt and L are respectively the *alphabet* and the *language* of pr. Please note that no further assumption is made on the language, namely wrt prefix-closure. In the sequel, we denote by **Pr** the collection of all such processes.

Object = behaviour + initiatives + state-dependent attributes

Given a process over an alphabet Evt of events, we obtain an object by endowing the process with an alphabet Att of *attributes* plus an *attribute valuation map* α providing their values after each trace of events. As an illustration of a (passive) object consider a "bank account" with the following alphabets:

$$\begin{aligned} Evt_{account} &= \{open, close\} \\ &\quad \cup \{withdrawal(i), deposit(i) : i \in integer\} \\ Att_{account} &= \{balance, red?\} \end{aligned}$$

For the sake of simplicity we are considering in these first examples that all attributes are integer valued (e.g. red?=1 means that the "account" is "in the red", that is, with negative balance), but the framework is easily established within a many-sorted data universe.

Clearly, in this setting events correspond to "atomic" *methods* as they are widely known within the community of object research. But we prefer to remain faithful to the classical terminology widely used by the community of process research, since we want to stress the behavioural aspects of objects and capitalize on the stable results of the theory of processes.

The behaviour of this object is some process over its event alphabet, possibly with the following language given as a regular expression:

$$L_{account} = open\ (withdrawal + deposit)^*\ close$$

The simple classical model of sequential processes that we are using does not recognize causality. But if we want to distinguish between active and passive objects it is important to enrich the behaviour model so that we know which is the object responsible for each event. This enrichment is simple — it is enough to partition Evt into two disjoint sets:

$$Evt = Init \cup Pass$$

The elements of Init are called *initiatives*. The elements of Pass correspond in many cases to *services*. The former are assumed to happen by the initiative of the object. The latter happen only on request from the environment of the object. As expected an object is said to be *passive* iff the set of its initiatives is empty. As an illustration of a passive object we have:

$Init_{account} = \varnothing$

$Pass_{account} = Evt_{account}$

As an example of an active object with initiatives consider now a "client" that may get the number 99:

Evt_{client} = {become_client_request, req_refusal, open_acc, close_acc}
∪ {become_client(i), withdrawal(i), deposit(i) : i∈ integer}

L_{client} = become_client_request req_refusal
+ become_client_request become_client(99)
open_acc (withdrawal + deposit)* close_acc

$Init_{client}$ = {become_client_request, open_acc, close_acc}
∪ {withdrawal(i), deposit(i) : i∈ integer}

$Pass_{client}$ = {req_refusal} ∪ {become_client(i) : i∈ integer}

Att_{client} = {number}

In conclusion, given a many-sorted algebra **Dt** of data-values, we define an *object* ob as a triple composed of a process, a subset of its alphabet of events, and an observation structure over that alphabet:

ob = ((Evt,L),Init,(Att,cod,α))

where

Init ⊂ Evt is the *alphabet of initiatives*;

(Att,cod,α) is the *attribute observation structure* over Evt, that is:

Att is a set (the *alphabet of state attributes*);

cod: Att →{Dt_σ: σ is a data-sort} (the *codomain map*);

$\alpha = (\alpha_a)_{a\in Att}$

where

α_a: Evt* → $2^{cod(a)}$

is the *valuation map for attribute* a, satisfying

$\alpha_a(\varepsilon) = \varnothing$.

For any (state) attribute a, the map α_a returns a set of values in the codomain of a for each finite sequence of events. Given such a sequence s of events, for each a∈ Att, if $\alpha_a(s)$ is empty we

say that the attribute a is *undefined* after the sequence s. It may also happen that an attribute is given more than one value (when α returns a non-singular set). Thus, wrt attribute values this model allows for *non-determinism.*

Naturally, we can look at the family $\alpha = (\alpha_a)_{a \in Att}$ as a map that for each sequence s of events returns the following set of pairs (attribute,value):

$$\alpha(s) = \{(a,d): a \in Att \wedge d \in \alpha_a(s)\}$$

In the sequel we shall work occasionally with this view of α (as the *overall valuation map* of the object):

$$\alpha : Evt^* \rightarrow Obs$$

where

$$Obs = 2^{\{(a,d): a \in Att \wedge d \in cod(a)\}}$$

is called the *observation set* of the object. We shall also need the extension of α to 2^{Evt^*} (by pointwise union).

As illustration of the valuation mechanism, consider the "account" and the "client" again (still assuming that all attributes are integer-valued):

$$\begin{aligned}
(\alpha_{balance})_{account}(open) &= \{1000\} \\
(\alpha_{red?})_{account}(open) &= \{0\} \\
(\alpha_{balance})_{account}(s\ deposit(i)) &= \{j+i: j \in (\alpha_{balance})_{account}(s)\} \\
(\alpha_{red?})_{account}(s\ deposit(i)) &= \{0: (j > -i) \wedge j \in (\alpha_{balance})_{account}(s)\} \\
&\quad \cup \{1: (j \leq -i) \wedge j \in (\alpha_{balance})_{account}(s)\} \\
(\alpha_{balance})_{account}(s\ withdrawal(i)) &= \{j-i: j \in (\alpha_{balance})_{account}(s)\} \\
(\alpha_{red?})_{account}(s\ withdrawal(i)) &= \{0: (j > i) \wedge j \in (\alpha_{balance})_{account}(s)\} \\
&\quad \cup \{1: (j \leq i) \wedge j \in (\alpha_{balance})_{account}(s)\}
\end{aligned}$$

$$(\alpha_{number})_{client}(s\ become\_client(99)\ r) = \{99\}$$

Clearly, at a given "time", the so called *state* of the object at that "time" is determined by the sequence of events that have already happened until then. Thus, we can talk about the values of the attributes at any given "time".

As an additional example, using the overall valuation map, consider the "stack" of integers as a (passive) object:

Evt_{stack}	=	{open, pop, close} $\cup$ {push(i) : i$\in$ integer}
L_{stack}	$\subset$	open (push + pop)* close
$Init_{stack}$	=	$\varnothing$
Att_{stack}	=	{top, empty?}
α_{stack} (open)	=	{(empty?,1)}
α_{stack} (s push(i))	=	{(empty?,0), (top,i)}
α_{stack} (s push(i) pop)	=	α_{stack} (s)

As expected, we would like to impose that a "pop" is possible only if the "stack" is not "empty" (that is, only if empty?=0). Thus,

$$L_{client} = \{s\in (open(push + pop)^*close): s_n=pop \Rightarrow (\alpha_{empty?})_{stack}(s_1 \ldots s_{n-1})=\{0\}\}$$

This object should be compared with the traditional abstract data-type "stack" — the elements of the data-type roughly correspond to the states of the object. Clearly, the notion of object is much richer. The object can suffer actions (events) that change its state (but retaining its identity). The corresponding data operations return other data elements. The object has state-dependent attributes that also correspond to data operations within the data-type setting. The notion of object-type goes a step further away from the data-type notion as will be seen in section 5.

3. WHAT IS OBJECT AGGREGATION ?

Like processes, objects may be combined into "larger" objects, that display the joint behaviour of the components. This operation is known as *aggregation* in the field of object-oriented programming. At the process level, it should correspond to parallel composition. Aggregation of *independent* (non-interacting objects) is rather simple, like in the case of processes. In the presence of interaction, aggregation is more complicated, also like it is for processes.

Object interaction is traditionally classified into two categories: *memory sharing* and *communication*. According to the proposed model, the former corresponds to *attribute sharing* and the latter either to *event sharing* or to *event calling*. Attribute sharing and event sharing are special cases of *object sharing* (as originally recognised in [12,13,14]) which, naturally, also

explains component sharing in general. Event calling appears as a more complex mechanism (see [31]). All these interactions, as well as inheritance mechanisms, are formally presented via the general notion of object morphism.

Object morphism

We have been experimenting with different versions of object morphisms [4,11,13,14], always keeping in mind that they should lead to colimits corresponding to aggregations. Following [4], an *object morphism* $h: ob_1 \rightarrow ob_2$ is a pair (h_E,h_A) composed of an *event map*

$$h_E: Evt_1 \rightarrow Evt_2$$

such that both the *behaviour morphism condition* (BMC)

- $h_E^{\leftarrow}(L_2) \subset L_1$ (+)

where

$h_E^{\leftarrow}(\{\varepsilon\}) = \{\varepsilon\}$
$h_E^{\leftarrow}(\{e_1\}) = perm(h_E^{-1}(e_1))$
$h_E^{\leftarrow}(\{e_1 \ldots e_n\}) = h_E^{\leftarrow}(\{e_1\}) \ldots h_E^{\leftarrow}(\{e_n\})$ for $n \geq 1$
$h_E^{\leftarrow}(S) = \bigcup_{s \in S} h_E^{\leftarrow}(\{s\})$,

and the *initiative morphism condition* (IMC)

- $h_E^{-1}(Init_2) \subset Init_1$

are satisfied;

and an *attribute map*

$$h_A: Att_1 \rightarrow Att_2$$

such that both the *codomain morphism condition* (CMC)

- $\forall a \in Att_1 \;\; cod_2(a) \subset cod_1(a)$ (++)

and the *valuation morphism condition* (VMC)

- $\forall s_2 \in Evt_2^*\ \ h_A^{\leftarrow}(\alpha_2(s_2)) \subset \alpha_1(h_E^{\leftarrow}(\{s_2\}))$ (+++)

where

$h_A^{\leftarrow}(\rho) = \{(a,i): a \in Att_1 \wedge (h_A(a),i) \in \rho\}$ for any $\rho \in Obs_2$

are satisfied.

Intuitively, the BMC imposes that the set of all events mapped by h_E into a single event of ob_2 may appear in any order (i.e. they are "concurrent") in ob_1. The IMC establishes that all initiatives of ob_2 must be initiatives of ob_1. The CMC allows smaller attribute codomains in ob_2. Finally, VMC imposes that the attributes of ob_1 mapped by h_A into ob_2 are evaluated in ob_2 precisely as they are in ob_1.

An object morphism h is said to be *full* whenever the inclusions (+), (++) and (+++) above degenerate in equalities. And it is said to be an *injection* (*inclusion*) when both h_E and h_A are injective maps (inclusions). In this case, we say that ob_2 *inherits* the behaviour, initiatives and attributes from ob_1. Thus, we envisage to formalize inheritance mechanisms using object inclusions.

The *category of objects* (that we shall denote by $\mathbb{O}b$) is established with the objects and the object morphisms we just introduced. In [4] it is shown that it is cocomplete and that the colimits do correspond to aggregations, as illustrated below.

The relationship *is-part-of*

We say that ob_1 *is-part-of* ob_2 when there is an inclusion morphism from ob_1 into ob_2. In this situation ob_1 is also said to be a *subobject* of ob_2.

We are now ready to characterize object *aggregation* (without and with interaction) using colimits in the category $\mathbb{O}b$, keeping in mind that wrt behaviour it should correspond to parallel composition. For instance, binary coproducts exist in $\mathbb{O}b$ and provide the disjoint aggregation of the argument objects: disjoint union of alphabet of events and disjoint parallel composition of behaviours (interleaving of processes), plus disjoint union of initiatives, plus disjoint union of alphabets of attributes and addition of valuation maps.

As an illustration consider the coproduct of two "bank accounts" — $account_1$ and $account_2$ — $account_1||account_2$. Note that we are extending the traditional notation for parallel composition of processes (||) to object aggregation, since they are so closely related. For instance, the event alphabet of the resulting aggregation contains two distinct "open" events, say $account_1$.open and $account_2$.open in a notation that shows their different origins. As expected these two independent objects can be put together without affecting each other. Clearly, for instance, $account_1$ *is-part-of* $account_1||account_2$.

Now consider some "client" owning some "account". We want to establish their joint behaviour in the presence of the usual interaction. Namely, the "account" is opened by the "client". Assuming that the objects account and client are as specified before, we would like to impose

client.open_acc = account.open
client.deposit(i) >> account.deposit(i)
client.withdrawal(i) = account.withdrawal(i)
client.close_acc = account.close

That is, "open", "close" and "withdrawal" actions are to be shared by the two objects. On the contrary, wrt "deposit" actions the interaction is much weaker — somebody else may also make "deposits" in the "account". Thus, we have two forms of interactions via events: (1) by *event sharing* (corresponding to a pushout in the category of objects, as already recognized in [11,12,13,14]); and (2) by *event calling* (corresponding to another colimit — for details see [31], although in a slightly different categorial setting).

The aggregation client||account displays the joint behaviour of these two interacting objects. Even in such a case of interacting components, each component *is-part-of* the whole. For instance, account *is-part-of* client||account .

Another form of interaction can be achieved by *attribute sharing*. However it should be stressed that, in order to obtain the envisaged joint behaviour, all the events affecting the attribute must also be shared. That is, it is only possible to share whole objects. This seems to be essential to object orientation. For details see for instance [13,14] and comments below.

Finally, it is interesting to analyse the import of the *is-part-of* morphism requirements. The behaviour morphism constraint (BMC) just states that every trace of account||client must degenerate into a legal trace of, for instance, account when we forget the other events — the basic requirement for interleaving.

The initiative morphism constraint (IMC) is also very natural. It imposes that the initiatives of the aggregation are the union of the initiatives of the parts. That is, an event of the aggregation is an initiative iff there is at least a component where it is an initiative.

The codomain morphism constraint (CMC) imposes that the codomain of an attribute in some aggregated object may be a subset of its codomain in a component. That is, we can further constrain a codomain when we go from the part to the whole.

The valuation morphism constraint (VMC) simply states that, within account||client, the values of the attributes imported from, say, account may depend only on the events imported from that part.

This constraint is also known as the *no collateral effects principle*, since it disallows the possibility of having a foreign event of the aggregated object affecting some attribute of a given part. Equivalently, this principle can be stated as imposing the sharing of all events changing some attribute whenever we want to share that attribute.

4. WHAT IS OBJECT SPECIALIZATION ?

At this point we are ready to discuss what is the meaning of ob_2 *is-a* ob_1, that is, ob_2 is a *specialization* of ob_1. Intuitively, we would like it to be: ob_2 can be taken as ob_1 if we forget a few details. For instance, a "savings_account" is an "account" in this sense. The former may have a few more attributes and events, more specific behaviour and valuation, but it should be possible to forget all that and recognize a simple "account".

The relationship *is-a*

We have been claiming [13,14] that (at least a strong version of) the *is-a* relationship corresponds to some object morphism (in the opposite direction). In this strong sense, ob_2 *is-a* ob_1 iff there is an inclusion morphism from ob_1 into ob_2.

That is, we say that

savings_account *is-a* account

iff there is an inclusion morphism from account into savings-account. Thus, iff account *is-part-of* savings_account! This strong version of the *is-a* relationship has been criticized in the past [23] and we have been experimenting with weaker versions [36].

Herein, we illustrate only the strong version:

$$Evt_{savings\_account} = \{open, close, start\_saving, new\_day, end\_saving\} \cup \{withdrawal(i), deposit(i) : i \in integer\}$$

$$Init_{savings\_account} = \varnothing$$

$$Att_{savings\_account} = \{balance, red?, saving?, days\}$$

The attribute days counts the number of "days" since the beginning of the saving period and is undefined when the "savings_account" is not in the saving mode. The attribute saving? indicates if the "account" is in the saving mode (saving?=1) or not (saving?=0).

With respect to constraining the behaviour, we might impose that if the "savings_account" is in the saving mode (that is, if saving?=1) then no "withdrawals" are allowed (only "deposits").

But, in order to illustrate another aspect discussed below, we prefer instead to allow an "withdrawal" to happen even when the "account" is in the saving mode and consider that in that event the mode is changed back to non-saving:

$$(\alpha_{saving?})_{savings\_account}(open) = \{0\}$$
$$(\alpha_{saving?})_{savings\_account}(s\ deposit(i)) = (\alpha_{saving?})_{savings\_account}(s)$$
$$(\alpha_{saving?})_{savings\_account}(s\ withdrawal(i)) = \{0\}$$
$$(\alpha_{saving?})_{savings\_account}(s\ start\_saving) = \{1\}$$
$$(\alpha_{saving?})_{savings\_account}(s\ end\_saving) = \{0\}$$
$$(\alpha_{saving?})_{savings\_account}(s\ new\_day) = (\alpha_{saving?})_{savings\_account}(s)$$

$$(\alpha_{days})_{savings\_account}(open) = \varnothing$$
$$(\alpha_{days})_{savings\_account}(s\ deposit(i)) = (\alpha_{days})_{savings\_account}(s)$$
$$(\alpha_{days})_{savings\_account}(s\ withdrawal(i)) = \varnothing$$
$$(\alpha_{days})_{savings\_account}(s\ start\_saving) = \{0\}$$
$$(\alpha_{days})_{savings\_account}(s\ end\_saving) = \varnothing$$
$$(\alpha_{days})_{savings\_account}(s\ new\_day) = \{j+1 : j \in (\alpha_{days})_{savings\_account}(s)\}$$

Note that the original events of "account" are allowed to change the "specific" attributes of "savings_account" — namely "withdrawals" may change the extra attributes. On the contrary, the valuation morphism condition (VMC) imposes that no specific event of "savings_account" may change an attribute of "account". This is a requirement of the strong version of *is-a* that seems to be rather restricting in many situations.

For instance, assume that we would like to have a fee (say of 50) being automatically taken from the "savings_account" whenever a saving period starts. It would *not* be possible to impose

$$(\alpha_{\text{balance}})_{\text{savings\_account}}(s\ \text{start\_saving}) \quad = \{j\text{-}50: j \in (\alpha_{\text{balance}})_{\text{savings\_account}}(s)\}$$

because in that case we would not have a morphism from account into savings_account. But the envisaged automatic fee can easily be imposed, still satisfying the morphism requirements, as follows:

start_saving >> withdrawal(50)

That is, the calling mechanism is rather useful when "collateral effects" are desired of specific events on non-specific attributes. Conceivably, a similar example arises around the automatic deposit of the interest at the end of the saving period.

With respect to behaviour, the morphism requirements are very simple and natural: any possible trace of "savings_account" should degenerate in a possible trace of "account" when we forget the specific events. Note again that we are free to impose further constraints on the behaviour of "savings_account" even with respect only to the non-specific events.

It should be stressed that the strong form of specialization that we have been discussing entails that the two objects are synchronized (at least) in all non-specific events — for instance, every "withdrawal" is an event of both objects ("savings_account" and "account").

Finally, it is worthwhile to discuss the morphism requirements on the initiatives. We may impose initiatives on "savings_account" as long as they do not correspond to non-specific passive events. That is, without destroying the morphism we could declare end_saving as an initiative, but not, say, withdrawal. This is a rather natural constraint that says that specializations inherit the passiveness of the events.

Method overriding

Even within the context of the strong version of specialization it is possible to understand what is known as "method overriding". In general, by overriding it is understood the redefinition of a method when inherited by a specialized object. In our context this would mean redefining an event.

We already saw an example of this mechanism: the withdrawal event of the account object is redefined to some extent in the savings_account object. Indeed, in the latter an "withdrawal" has *more effects* than in the former — namely, it also affects the values of the saving? specific attribute.

The rather strict form of specialization we introduced using the notion of object morphism does allow for *event overriding* in this restricted form: an event when inherited may have further effects, but the effects on the inherited attributes must remain the same (or, at most, be less liberal). We could say that we have here only "monotonic" overriding.

Any weaker (more liberal) form of specialization will also lead to a more liberal overriding mechanism allowing, for instance, a different treatment of the inherited attributes by the inherited events in the specialized object.

5. WHAT IS AN OBJECT-TYPE ?

So far we have been working only with single objects. Now we consider collections of objects belonging to the same "type". Basically, an *object-type* is a pair:

$$ot = (Id, \omega)$$

where Id is the *identification space* (or *surrogate space*) of ot — a carrier set for the underlying data algebra; and ω is the *template map* of ot — a map from Id into |Obl. In this context, for each $x \in Id$ we have: $\omega(x)$ is the *template* of x; and $x.\omega(x)$ is the object *instance* of ot corresponding to x.

An object-type is said to be *homogeneous* when ω is constant. As an illustration consider the object-type ACCOUNT defined as follows:

$$Id_{ACCOUNT} = \{acc(i): i \in integer\}$$

$\omega_{ACCOUNT}(x)$ = account (as defined in section 2)

Alternatively, we might want to number "accounts" locally to each "branch" of the bank. In that case we would have:

$Id_{ACCOUNT}$ = {acc(b,i): b∈ Id_{BRANCH} ∧ i∈ integer}
$\omega_{ACCOUNT}(x)$ = account (as defined in section 2)

It is beyond the scope of this paper to elaborate on useful, typical ways of constructing identification spaces for the relevant object types. Further examples and methodological guidelines can be found in [33,35].

Data modeling

When using objects for *conceptual modeling* the choice of types and their surrogate spaces and relationships corresponds to the traditional task of "data modeling". But with objects, conceptual modeling is extended to other aspects of the real-world entities, namely their dynamic behaviour. The role of dynamic modeling within the setting of an object-oriented approach to systems development is illustrated in some detail in [33].

In any case, it should be clear even from the incipient examples above that the choice of the relevant object types starts with a task akin to classical "data modeling". Indeed, the construction of the identification spaces of the relevant object types establishes the interesting "entities" and their "relationships".

It has been argued in the past that a pure object-oriented approach would establish a specialization hierarchy with a very general object type at the top, for which it would be meaningless to try to find an interesting identification space [28]. However, our practical experience in object-oriented development of *information systems* and their *databases* suggests the opposite! The traditional "data modelling" is still relevant mainly in the early stages of the hunt for the relevant "objects" and their "relationships".

Specialization at the type level

We are now ready to discuss the inheritance mechanisms at the object-type level. A significant part of such mechanisms can be formalized via the general notion of *object-type morphism* that also serves to explain the relationship between complex object-types and their component types

(for details see [13]). Herein, we shall illustrate the notion of object-type morphism only for the simplest case that corresponds to subtyping — that is, only for type specialization.

An object-type ot_2 is said to be a *subtype* of ot_1 iff $Id_2 \subset Id_1$ and, for each $x \in Id_2$, $x.\omega_1(x)$ is a subobject of $x.\omega_2(x)$. In this case, we say that each instance of ot_2 *inherits* the behaviour, initiatives and attributes from the corresponding instance of ot_1.

As an illustration take the homogeneous subtype SAVINGS_ACCOUNT of the homogeneous object-type ACCOUNT, possibly having more attributes (say number of saving days) and more events (say starting and ending saving period), as well as more restricted behaviour. Naturally, the ACCOUNT attributes and events are inherited, as well as the valuation map between them.

6. WHAT IS OBJECT IMPLEMENTATION ?

Although that has not been widely recognized so far, it seems essential to a full understanding of the object-oriented computation paradigm the availability of an effective theory of object implementation. Unfortunately, such a theory will have to be closer to a theory of process implementation than to a theory of abstract data-type implementation. The latter is by now well established [8,9,15,20,30], but with respect to processes the theory of implementation is still on its infancy.

One of the main reasons why process implementation is a non-trivial matter is because it may involve a change of "granularity" — an event may be implemented as a "complex" action. This action nonetheless must keep its "atomicity", in the sense that either it does not happen (when the event does not happen) or it is carried out to the end (when the event does happen). Such "complex" actions with such "transactional requirements" are called "transactions".

Thus, before we consider object implementation, we should analyze the process model adopted in section 2 in order to see how the notion of transaction fits into it.

Processes with transactions

Liveness commitments such as those associated to transactional behaviour (of the type "carry out until the end") can easily be described within the proposed model without any further enrichment. Indeed, we can represent *transactional requirements* by not including in the language the prefixes of each "transaction". For instance, the process $\{\varepsilon, e_1, e_1e_2, e_2e_3, e_2e_3e_1\}$

over $\{e_1,e_2,e_3\}$ may start with a transactional requirement: if it engages in e_2 it must perform e_3 since the prefix e_2 of e_2e_3 does not belong to the language. That is, the process may start with $<e_1>$ or $<e_2e_3>$. Sequences as these ones are called "transactions".

Formally, following [5], the alphabet Trans of *transactions* of a process pr = (Evt,L) is obtained as follows:

$$Trans = \{<s>: s \in Evt^+\}$$

The *transactional language* TL of a process pr = (Evt,L) is derived as follows:

$$TL = \varphi(L)$$

where the *process folding map* φ is recursively calculated as follows:

$$\varphi(\{\varepsilon\}) = \{\varepsilon\}$$
$$\varphi(S) = \{<s>r: s \in \tau(S) \wedge r \in \varphi(S/s)\}$$

assuming that $\tau(S)$ (the *transactional menu* of S) and S/s (S *after* s) are as follows:

$$\tau(S) = \{s \in S: s \neq \varepsilon \wedge \forall\, r \leq s\ (r \in S \Rightarrow (r=\varepsilon \vee r=s))\}$$
$$S/s = \{r: sr \in S\}$$

In the example above we obtain $TL = \{\varepsilon,<e_1>,<e_1><e_2>,<e_2e_3>,<e_2e_3><e_1>\}$ well illustrating the following general result: TL is prefix-closed.

A process pr = (Evt,L) is said to be *transaction-free* iff L is prefix-closed. Otherwise pr is said to have transactional requirements. In any case φ(pr) denotes the *folded process* obtained by "folding" the transactions of pr into events:

$$\varphi(pr) = (Trans,TL)$$

By using the process folding map φ we can derive from any object

$$ob = ((Evt,L),Init,(Att,cod,\alpha))$$

the corresponding *folded object*

$\varphi(ob) = (\varphi((Evt,L)),\varphi(Init),(Att,cod,\varphi(\alpha)))$

where $<e_1 \ldots e_n> \in \varphi(Init)$ iff $e_1 \in Init$; and $\varphi(\alpha)(s <e_1 \ldots e_n>) = \alpha(s\ e_1 \ldots e_n)$.

Implementing an object over a given base

With these concepts we are at last ready to tackle the problem of implementing an object (the "abstract" object *abs*) over another object (the "base" object *bas*), possibly using transactions (broadly following [5,11]). Clearly, the attributes of *abs* should be obtained from the attributes of *bas* and the events of *abs* should correspond to transactions built with events of *bas*. This construction of *abs* with *bas* is achieved via a third object (the "middle" object *mid*) as follows.

First *mid* is obtained by enriching *bas* with the necessary derived attributes and transactional requirements (thus establishing an inclusion morphism from *bas* into *mid*). This morphism is also known as an *enrichment morphism*. Then we have to verify that there is a *reification morphism* from *abs* into *mid* — that is, a (trace-dependent) "encoding" of the *abs* events in terms of the *mid* transactions. The existence of this morphism is the *correctness condition* for the envisaged implementation.

As an illustration consider the popular example of the implementation of a "stack" over the aggregation of an "array" and a "pointer". Please note that all these objects have deterministic attributes.

For instance, the attribute top of *mid* is to be derived from the contents of the array (conts) at the position indicated by the pointer (val) minus 1; and the event push(i) of stack is to be mapped into the transaction that stores i at the position indicated by the pointer and afterwards increments the value of the pointer: <set(val,i) asg(val+1)>. That is, the pointer is assumed to indicate at all times the next free position in the array. When the "stack" is empty the pointer has value 0.

Clearly, we are assuming the following components of the base:

Evt_{array}	=	{new, drop} ∪ {set(n,m) : n,m∈ integer}
L_{array}	=	new (set)* drop
$Init_{array}$	=	∅
Att_{array}	=	{conts(n): n∈integer}

α_{array} (new) = $\varnothing$

α_{array} (s set(n,m)) = {(conts(i),j) : i≠n ∧ j∈ ($\alpha_{conts(i)}$)$_{array}$ (s)} ∪ {(conts(n),m)}

$Evt_{pointer}$ = {alloc, free} ∪ {asg(n) : n∈integer}

$L_{pointer}$ = alloc (asg)* free

$Init_{pointer}$ = $\varnothing$

$Att_{pointer}$ = {val}

$\alpha_{pointer}$ (alloc) = $\varnothing$

$\alpha_{pointer}$ (s asg(n)) = {(val,n)}

Thus, the base object *bas* is array||pointer (aggregation of the two components). The middle object *mid* is to be established as a suitable enrichment of *bas* along the ideas given above, so that we can later on "encode" the abstract object stack:

Evt_{mid} = Evt_{array} ∪ $Evt_{pointer}$

$Init_{mid}$ = $\varnothing$

Att_{mid} = Att_{array} ∪ $Att_{pointer}$ ∪ {top, empty?}

$(\alpha_{empty?})_{mid}$ (s) = if $(\alpha_{val})_{mid}$ (s)={0} then {1} else {0}

$(\alpha_{top})_{mid}$ (s) = if $(\alpha_{val})_{mid}$ (s)={0} then $\varnothing$ else $\bigcup_{n\in(\alpha_{val})_{mid}(s)}(\alpha_{conts(n-1)})_{mid}$ (s)

Clearly, the stack attributes are "derived" from the attributes imported from the base. It remains to impose the envisaged trace-dependent "encoding" ρ of the stack events in terms of the transactions of *mid* and to restrict the traces of *mid* to those "encoding" traces of stack:

ρ: $(Evt_{stack})^*$ Evt_{stack} → $Trans_{mid}$

From this (trace-dependent) *event encoding map* we can derive the corresponding *trace encoding map*

$R: (Evt_{stack})^* \rightarrow Trans_{mid}^*$

as follows:

$R(\varepsilon) = \varepsilon$

$R(s\ e) = R(s)\ \rho(s,e)$

In the example at hand we want to impose:

$\rho(s,open)$	=	<new alloc asg(0)>
$\rho(s,close)$	=	<drop free>
$\rho(s,pop)$	=	<asg(n-1)> provided that $(\alpha_{val})_{mid}(R(s))=\{n\}$
$\rho\ (s,push(i))$	=	<set(n,i) asg(n+1)> provided that $(\alpha_{val})_{mid}(R(s))=\{n\}$

It is hardworking but almost straightforward to verify that indeed we achieved an implementation of the stack with this object *mid*, in the sense that the allowed behaviour of the latter is the full "encoding" of the behaviour of the former and at each state we can "derive" the stack attributes from the other attributes. The relationship thus established between stack and *mid* corresponds to what we call a reification morphism. In general, this relationship is even more complex because the object *abs* may already have transaction requirements that have to be fulfilled by the implementation. Also in the example we forgot about initiatives because the objects at hand were all passive.

Formally, still following [5], we say that there is a *reification morphism*

$\rho: ob_1 \rightarrow ob_2$

iff

$\rho = (\rho_T, \rho_A)$

composed of an injective *transaction map*

ρ_T: $Trans_1^*$ $Trans_1 \rightarrow Trans_2$

such that

$\rho_T(s,<t_1 \ldots t_n>) = <\rho_T(s,t_1) \ldots \rho_T(s\ t_1 \ldots t_{n-1},t_n)>$,
for every $s \in Trans_1^*$, $t_1,\ldots,t_n \in Trans_1$

$L_2 = R_T(L_1)$

where R_T: $Trans_1^* \rightarrow Trans_2^*$ is derived as follows:
$R_T(\varepsilon) = \varepsilon$
$R_T(s\ t) = R_T(s)\ \rho_T(s,t)$

$\rho_T(s,t) \in \varphi(Init_2)$ iff $t \in \varphi(Init_1)$, for every $s \in Trans_1^*$, $t \in Trans_1$

and an injective *attribute map*

ρ_A: $Att_1 \rightarrow Att_2$

such that

$cod_2(a) = cod_1(a)$, for every $a \in Att_1$

$\alpha_2(R_T(s)) = \alpha_1(s)$, for every $s \in Evt_1^*$

In this situation we say that ob_2 is a reification (or encoding) of ob_1. Note that barring the possibility of granularity change (an event of ob_1 mapped into a transaction of ob_2) it corresponds to a special case of object morphism (injective and full).

Clearly, given an object *abs* and a target base *bas* it may well be impossible to find a suitable enrichment *mid* of *bas* in which to encode *abs*.

This rather involved notion of object reification seems to be a minimal one in the setting of the process model we adopted. In [5] both horizontal and vertical compositionality issues are

addressed, namely how we can compose nested implementations and how we can combine implementations of parts in order to obtain an implementation of the whole.

Traditional implementation platforms

The practical import of object reification issues in systems development should be obvious. In addition, we have to understand the elements of the traditional implementation platforms (e.g. database relations and application procedures) as objects. In this way the conceptual clash between object-orientation and traditional implementation architectures is minimized. However, efficiency problems arising from this mismatch are yet to be analysed.

Clearly, for instance, a *database relation schema* R can be taken as defining an object-type. Each tuple of the relation in the database will correspond to an existing instance of the object-type. The relation itself (set of tuples) corresponds to the class (in the object-oriented terminology). But going beyond the relational framework, besides key attributes (used for constructing the identification space) and state attributes, the definition of the object-type R also includes events (atomic methods) for accessing and manipulating the tuples of R. Thus, within the object-oriented setting we can endow relations with the associated procedures and transactions. Furthermore, we can also discuss the intended behaviour of the (passive) tuples of R, for example in terms of which "manipulation operations" are enabled at each state.

On the other hand, each run of an application program can also be taken as an instance of an object-type. That is, each application program (or transaction) P can be taken as an object-type. Obviously, the instances of P are non-passive objects, usually transient ones. As an illustration consider the application CALCULATE_INTEREST to be triggered (that is, born) say whenever the mode of some "savings_account" is changed back to non-saving. The state attributes of CALCULATE_INTEREST will correspond to the fields (or variables) of the work-area. Interaction with the data-base relations (reading and writing) is also easily understood as object interaction of one form or another.

It should be stressed however that the traditional classification of objects into these two disjoint families (on one hand, the passive, persistent objects to be stored in the data-base; and, on the other hand, the active, usually transient objects to be coded in some programming language and managed at run time by the transaction manager and/or the operating system kernel) is motivated only by current implementation technology (systems and languages). It seems that a better way would be to concentrate instead most of the activity of the system around the objects that until now are taken as passive and that in current systems are stored in the database.

Finally, we should also refer *en passant* the import of the object paradigm in the area of human-machine interaction (HMI). It is well beyond the scope of this paper to elaborate on this, but it is now clear that HMIs are better understood and designed as objects. For an example based on the concepts introduced herein see [38].

7. TOWARDS OBJECT DESIGN LANGUAGES AND CALCULI ?

In the previous sections we outlined the semantic foundations of object-orientation, including many of its aspects (inheritance, types and reification). We believe that such "mathematical semantics" is essential to a better understanding of the object-oriented paradigm of computation.

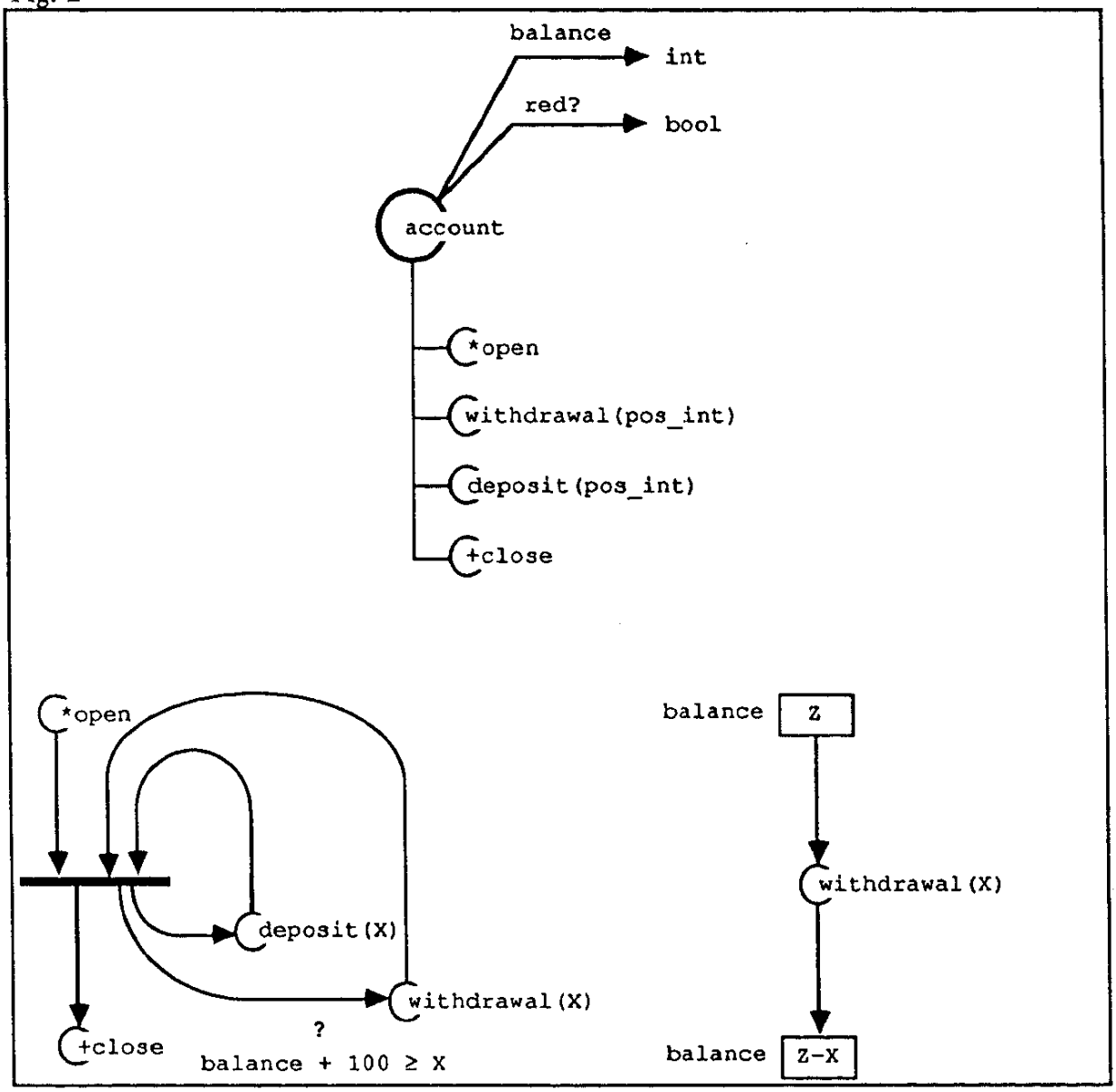
Fig. 1

```
object account
  attributes
    balance : int ;
    red? : bool
  events
    * open ;
    withdrawal(pos_int) ;
    deposit(pos_int)
    + close
  behaviour
    { balance + 100 ≥ X } withdrawal(X)
  valuation
    [open]balance = 1000 ;
    [open]red? = false ;
    [withdrawal(X)]balance = balance - X ;
    [withdrawal(X)]red? = if (X > balance) then true else false ;
    [deposit(X)]balance = balance + X ;
    [deposit(X)]red? = if (X > (- balance)) then false else true
end
```

However, the proposed semantic domain for objects and their types that we have outlining can only become useful if developed into a broad set of *languages*, *calculi* and *methods* for systems

specification, implementation and verification. Then (and only then!) a uniform, object-oriented approach to systems development will come into being.

Fig. 2



Let us consider herein only the special case of *information* systems. We would like to sketch suitable languages, calculi and methods supporting information system development, from the early stages of requirements definition, through conceptual modeling, to the later stages of layered implementation, towards a given target platform (say database/transaction management system), and, subsequently, also supporting system tuning and maintenance.

Ideally, we would like to have a single language to be used throughout the entire life-cycle of the information system. However, current trends in software engineering and in CASE suggest

that we will need two versions of the envisaged language: *textual* and *graphical*. As an illustration consider two equivalent specifications of an "account": textual (as given in Fig. 1) and graphical (as partially depicted in Fig. 2). Compared to the "account" discussed in section 2, we added here an enabling constraint on "withdrawals", assumed deterministic attributes, and adopted the most suitable codomains for them.

Clearly, we envisage a single language with two alternative presentations (textual and graphical). Thus, in the sequel we disregard such presentation issues and discuss only the single, underlying language.

Besides providing the means for defining single objects (as illustrated in Figs. 1 and 2), the language should also allow the definition of object-types, supporting the detailed construction of the object identification spaces according to suitable "data modeling" principles. This is of utmost importance in the context of *information* system development. Indeed, as we already pointed out in section 5, when developing information systems, objects should be classified into types providing instance identification mechanisms that are meaningful to the users. For instance, it is important to know if "accounts" are to be numbered locally in each "branch" of the bank or not. Since there is a long established tradition of conceptual languages satisfying this requirement, we shall not discuss further this topic.

Another dimension of the problem (for instance at the object level) is to make the language sufficiently versatile so that it is possible to specify with equal ease: (a) *requirements* (such as, the balance may not go lower than some limit) even before we commit ourselves to the event alphabet of the relevant objects; and (b) *behaviour* and *valuation* (as illustrated for account in Figs. 1 and 2). Requirements are easily specified with temporal and deontic constructs on the alphabet of attributes (see for instance [17]), but we refrain to illustrate them herein.

Moreover, this brings in the first example of verification — we would wish to have a logical calculus for verifying if the specified behaviour and valuation complies with the stated requirements. Such a reasoning can be made in many cases locally to an object: for instance when verifying if the behaviour and valuation given in Figs. 1 or 2 complies with the requirement:

```
always balance ≥ -100
```

But since objects do interact, in other cases it may be necessary to take into account other objects. That is, besides a *local* calculus for reasoning about a given object in isolation, we also need a *global* calculus for reasoning about interacting objects, as analysed in [16,18]. Clearly,

interesting compositionality problems are immediately raised — for instance, can we compose local theories into global theories? Such issues are addressed in [17]. Without going into details, we can say that not every assertion valid locally will remain valid when the object is embedded in a collection of interacting objects. Thus, it becomes interesting to find a local calculus that will have as theorems only those assertions that will remain valid in any context.

Another interesting aspect is inheritance, both wrt the language and the calculi. There is a long established tradition of object-oriented programming languages providing adequate constructs for inheritance. Thus, besides stressing the need to analyse carefully the interplay between such inheritance facilities and the other constructs, namely definition of behaviour and classification into types (with identification mechanisms), we shall not dwell further on the topic. On the other hand, with respect to the calculi, there is preciously little experience reported in the literature (see for instance [16]).

Finally, we have to consider reification, especially using transactions. Although we can capitalize somewhat on the ADT languages and calculi, object reification is closer to process reification and there is little experience on the latter. There is no difficulty in finding reasonable language constructs for expressing reification morphisms, once we have the transaction construct in the language. But verifying the correctness of an implementation implies a further extension to the calculi we have been discussing [17].

Clearly, the idea is to define the "mathematical semantics" of these language and calculi over the semantic domain for objects that we outlined in the previous sections. For the language that task is progressing without surprises, but with respect to the calculi we discovered that the suitable interpretation structures did not coincide with the proposed semantic domain (see for instance [17] and [19]). Indeed, only recently the relationship between the calculi interpretation structures and the category of objects (as introduced in section 3) has become clear, since the former deal with "embedded" objects (so that local theorems can be taken as valid assertions about aggregations of interacting objects) and the latter contains "isolated" objects. That is, the category of "embedded" objects (the interpretation structures of the calculi) appears as a comma category of the "isolated" objects (the elements of the proposed semantic domain).

8. CONCLUDING REMARKS

We outlined a simple model for concurrent objects displaying deterministic behaviour, either active or passive. Objects appear as processes endowed with initiatives and attributes. Although we used a specific process model, it should be clear that the object concept can be

lifted to other process models as well (e.g. allowing for non-deterministic processes). Object aggregation, inheritance mechanisms, object-types and reification techniques (with transactions) were easily described within the proposed model.

These semantic foundations of the object-oriented paradigm of computation have been under intensive research within the IS-CORE project (ESPRIT-2 BRA 3023). Beyond the model outlined herein, a most general framework was introduced in [10]. Also interesting are the fragmentary results, including animation aspects, on the trinity (denotational, operational and axiomatic semantics) of object behaviour [3].

We also discussed the practical import of the proposed model for objects, namely with respect to information system development languages and verification calculi. Such languages and calculi are also under intensive research within the IS-CORE project, as reported for instance in [17,19].

In conclusion, we should stress once more that an object-oriented approach to information system (and database) development should tackle the following two issues (that tend to be disregarded in the literature within the area of object-oriented systems and languages): (1) a typing mechanism for objects supporting the traditional "data modeling" concept of object identification; (2) a reification mechanism for objects supporting the notion of "transaction".

Only an effective treatment of these two aspects will make possible the object-oriented design of information systems to be implemented over currently available platforms (database/transaction managers). Furthermore, we believe that those "data modeling" and "transaction" concepts will survive beyond the use of the traditional implementation platforms because they are, after all, implementation independent.

ACKNOWLEDGMENTS

Part of this work has been carried out by the authors within the ESPRIT-2 BRA 3023 IS-CORE (Information Systems COrrectness and REusability), namely as part of task 1 (semantic fundamentals). The authors are greatly indebted to the other members of IS-CORE, especially José-Félix Costa, Francisco Dionísio and those also working in tasks 2 and 3 (languages and calculi respectively): Cristina Sernadas, José Fiadeiro, Gunter Saake, Tom Maibaum, Ralf Jungclaus and Udo Lipeck. The methodological import of the proposed semantic domain was also extensively discussed with and criticized by those members also working in task 4 (methods): Robert Meersman, Olga DeTroyer and Pedro Resende.

REFERENCES

[1] America, P.: Object-oriented Programming: A Theoretician's Introduction. In: *EATCS Bulletin*, **29**, pp 69-84, 1986.

[2] Atkinson, M., Bancilhon, F., DeWitt, D., Dittrich, K., Maier, D. and Zdonik, S.: The Object-oriented Database System Manifesto. In: *First International Conference on Deductive and Object-oriented Databases*. W. Kim, J.-M. Nicolas and S. Nishio (eds), pp 40-57, 1989.

[3] Costa, J.-F.: *Teoria Algébrica dos Processos Animados*, MSc Thesis, Universidade Técnica de Lisboa, September 1989.

[4] Costa, J.-F., Dionísio, F., Sernadas, A. and Ehrich, H.-D.: *The Ultimate (Finitely) Cocomplete Category of Sequential Processes*. INESC, Lisbon, 1990.

[5] Costa, J.-F., Sernadas, A. and Ehrich, H.-D.: Object Reification within a Cocomplete Category of Sequential Objects. INESC, Lisbon (in preparation).

[6] Dahl, O.-J., Myhrhaug, B. and Nygaard, K.: *SIMULA 67, Common Base Language*. Norwegian Computer Center, 1967.

[7] Dayal, U. and Dittrich, K. (eds): *Proceedings of the International Workshop on Object-Oriented Database Systems*. Los Angeles, IEEE Computer Society, 1986.

[8] Ehrich, H.-D.: On the Theory of Specification, Implementation and Parameterization of Abstract Data Types. In: *Journal of the ACM*, **29:1**, pp 206-227, 1982.

[9] Ehrich, H.-D., Gogolla, M. and Lipeck, U.: *Algebraische Spezifikation Abstrakter Datentypen*. B.G. Teubner, 1989.

[10] Ehrich, H.-D., Goguen, J. and Sernadas, A.: A Categorial Theory of Objects as Observed Processes. In: *Proceedings of the REX Workshop on Foundations of Object-oriented Languages*, Springer Verlag (in preparation).

[11] Ehrich, H.-D. and Sernadas A.: Algebraic Implementation of Objects over Objects. In: *Stepwise Refinement of Distributed Systems: Models, Formalisms, Correctness*, J. de Baker, W.-P. de Roever and G. Rozenberg (eds), LNCS 430, pp 239-266, Springer Verlag, 1990.

[12] Ehrich, H.-D., Sernadas, A. and Sernadas, C.: Abstract Object Types for Databases. In: *Advances in Object-Oriented Database Systems*, K. Dittrich (ed), pp 144-149, LNCS 334, Springer Verlag, 1988.

[13] Ehrich, H.-D., Sernadas, A. and Sernadas, C.: Objects, Object Types and Object Identity. In: *Categorical Methods in Computer Science with Aspects from Topology*, H. Ehrig et al (eds), LNCS 393, pp 142-156, Springer Verlag, 1989.

[14] Ehrich, H.-D., Sernadas, A. and Sernadas, C.: From Data Types to Object Types. In: *Journal of Information Processing and Cybernetics*, EIK 26(1/2), pp 33-48, 1990.

[15] Ehrig, H., Kreowski, H.-J., Mahr, B. and Padawitz, P.: Algebraic Implementation of Abstract Data Types. In: *Theoretical Computer Science*, **20**, pp 209-263, 1982.

[16] Fiadeiro, J.: *Cálculo de Objectos e Eventos*, PhD Thesis, Universidade Técnica de Lisboa, January 1989.

[17] Fiadeiro, J. and Maibaum, T.: Describing, Structuring and Implementing Objects. In: *Proceedings of the REX Workshop on Foundations of Object-oriented Languages*, Springer Verlag (in print).

[18] Fiadeiro, J. and Sernadas, A.: Logics of Modal Terms for Systems Specification. *Journal of Logics and Computation* (in print).

[19] Fiadeiro, J., Sernadas, C., Maibaum, T. and Saake, G.: Proof-theoretic Semantics of Object-oriented Specification Constructs. In: *Object Oriented Databases: Analysis, Design and Construction*, R. Meersman and W. Kent (eds.), North Holland (in print).

[20] Goguen, J., Thatcher, J. and Wagner, E.: An Initial Algebra Approach to the Specification, Correctness and Implementation of Abstract Data Types. In: *Current Trends in Programming Methodology, Vol. 4: Data Structuring*, R. T. Yeh (ed.), pp 80-149, Prentice-Hall, 1978.

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

[22] Goldblatt, R.: *Topoi, the Categorial Analysis of Logic.* North-Holland, 1979.

[23] Guendel, A.: *Compatibility Conditions on Subclasses.* University of Dortmund, 1990.

[24] Hennessy, M.: *Algebraic Theory of Processes.* The MIT Press, 1988.

[25] Hewitt, C.: Viewing Control Structures as Patterns of Passing Messages. In: *Journal of Artificial Intelligence*, **8:3**, pp 323-364, 1987.

[26] Hewitt, C. and Baker, H.: Laws for Communicating Parallel Processes. In *Proceedings of the 1977 IFIP Congress*, pp 987-992, IFIP, 1977.

[27] Hillis, W.: *The Connection Machine.* The MIT Press, 1986.

[28] Khoshafian, S. and Copeland, G.: Object Identity. In: *ACM SIGPLAN Notices*, **21:11**, pp 406-416, 1986.

[29] Lochovski, F. (ed): Special Issue on Object-Oriented Systems. In: *IEEE Database Engineering*, **8:4**, 1985.

[30] Sannella, D. and Tarlecki, A.: Toward Formal Development of Programs from Algebraic Specifications: Implementations Revisited. In: *Acta Informatica*, **25**, pp 233-281, 1988.

[31] Sernadas, A., Ehrich, H.-D. and Costa, J.-F.: From Processes to Objects. In: *The INESC Journal of Research and Development*, **1:1**, pp 7-27, 1990.

[32] Sernadas, A., Fiadeiro, J., Sernadas, C. and Ehrich, H.-D.: Abstract Object Types: A Temporal Perspective. In: *Proceedings of the Colloquium on Temporal Logic and*

Specification, B. Banieqbal, H. Barringer and A. Pnueli (eds), pp 324-350, Springer Verlag, 1989.

[33] Sernadas, A., Fiadeiro, J., Sernadas, C. and Ehrich, H.-D.: The Basic Building Blocks of Information Systems. In: *Information Systems Concepts*, E. Falkenberg and P. Lindgreen (eds), pp 225-246, North Holland, 1989.

[34] Sernadas, A., Sernadas, C. and Ehrich, H.-D.: Object-Oriented Specification of Databases: An Algebraic Approach. In: *Proceedings of the 13th VLDB Conference*, P. Hammersley (ed), pp 107-116, VLDB, 1987.

[35] Sernadas, C., Fiadeiro, J. and Sernadas, A.: Object-oriented Conceptual Modeling from Law. In: *The Role of Artificial Intelligence in Databases and Information Systems*, R. Meersman, Z. Shi and C.-H. Kung (eds), pp 305-327, North Holland, 1990.

[36] Sernadas, C., Saake, G. and Sernadas, A.: *Algebraic Approach to Inheritance.* INESC, Lisbon, 1990.

[37] Shriver, B. and Wegner, P. (eds): *Research Directions in Object Oriented Programming.* MIT Press, 1987.

[38] Sousa, J.-P., Sernadas, C. and Sernadas, A.: An Object-oriented Specification Toll for Graphical Interfaces. In: *Computer Graphics*, **14:1**, pp 29-40, 1990.

[39] Wieringa, R.: Equational Specification of Dynamic Objects. In: *Object Oriented Databases: Analysis, Design and Construction*, R. Meersman and W. Kent (eds.), North Holland (in print).