J. W. deBakker, W. P. deRoever, and G. Rozenberg, editors, Proc. REX/FOOL Workshop, LNCS 489, pages 203–228, Berlin, 1991. Springer

A Categorial Theory of Objects as Observed Processes

H.-D. Ehrich

Abteilung Datenbanken, Technische Universität Braunschweig, D-3300 Braunschweig, FR GERMANY

J. A. Goguen

Programming Research Group, Oxford University Computing Lab, Oxford OX1 3QD, GREAT BRITAIN

A. Sernadas

Departamento de Matematica, Instituto Superior Técnico, 1096 Lisboa Codex, PORTUGAL

Abstract - *The semantic domain for object-oriented languages and systems outlined in this paper is an amalgamation of two approaches: the objects-as-sheaves approach of the second author published nearly two decades ago, and the more recent ideas of the first and third authors on objects as processes endowed with observation. The basic insight is that objects in the latter sense correspond to object morphisms in the former sense. After an informal introduction into both approaches, we first elaborate on the sheaf model, using the term "behaviour" for objects in this sense, and avoiding concepts from topology. Behaviours and behaviour morphisms are shown to form a complete category where parallel composition is reflected by limits. Objects are defined to be behaviour morphisms, giving rise to a cocomplete category of objects where colimits reflect object aggregation. Object morphisms reflect different forms of inheritance, and also object reification (implementing objects over objects) is conveniently expressed in this framework.*

Key words - object-oriented system ; object ; object morphism ; process ; behaviour ; object aggregation ; parallel composition ; object inheritance ; object reification.

1. Introduction

What is an object? Although substantial agreement has been obtained on many basic intuitions, as yet there is no coherent theory which can cope with all aspects, including object interaction and aggregation, object inheritance, object types and classes, object specification and implementation, object correctness and verification, etc., and which can provide a sufficiently rich and reliable basis for designing, implementing and using object-oriented languages and systems.

It is standard to view object-oriented systems as communities of interacting objects where all objects operate concurrently on data of various types. Accordingly, process theory and abstract data type theory provide relevant building blocks for object theory, but their integration is far from trivial. There are many different formalisms, and it is difficult to compare, combine or apply them. In particular, logics and models are often not clearly distinguished, and are rarely combined. Moreover, there are many different levels of abstraction.

This paper combines two semantic approaches to object theory. It restructures the objects-as-observed-processes approach developed mainly by the first and third authors in view of the objects-as-sheaves approach of the second author, first published nearly two decades ago.

Sheaf theory developed in mathematics for studying relationships between local and global phenomena, and has been applied in algebraic geometry, differential geometry, analysis, and even logic. It has also been developed in an abstract form using category theory (Gra65, Gro71). Section 2.2 gives an informal overview of this approach, and full information can be found in (Go71, Go75, Go90a).

Section 2.1 reviews the basic ideas of the objects-as-observed-processes approach. Its development can be traced in (SSE87, SFSE89a, SFSE89b, ESS89, ESS90, ES90). The main difference from previous papers is the uniform treatment of processes and observations influenced by the sheaf approach: both the process part and the observation part appear as "objects" in the latter sense, called "behaviours" here in order to avoid confusion. These parts are related by a behaviour morphism which tells how the process "triggers" observations.

The mathematics of behaviours and behaviour morphisms is developed in chapter 3 in a purely categorial framework, establishing the category BHV of behaviours. BHV is shown to be complete, and limits are shown to reflect parallel composition of behaviours.

In chapter 4, we introduce objects as behaviour morphisms, and object morphisms as commutative squares in BHV. This way, the category OB of objects is constructed from BHV by a well known categorial construction, namely as a "comma category". OB is shown to be cocomplete, with colimits reflecting object aggregation. Our very general notion of object morphism is shown to cover different kinds of inheritance relationships between objects as special cases. Finally, we briefly describe how object reification (implementing objects over objects, cf. ES90) is conveniently expressed in this framework.

2. Motivation

2.1 Objects as Observed Processes

Following the argument in SE90, a computer system as a whole is a symbolic machine which is able to receive, manipulate, store, produce and transmit data. 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 instance of either one of them is created and evolves through time (i.e. changes its state), possibly moving from one place to another, until it is finally destroyed (if ever). Any such instance is able to retain data, is able to replace the data it retains, and may be either 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 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.

In conclusion, barring the liveness and initiative issues, an object (or actor as some authors prefer to call it when a community of full concurrent objects is involved) is a process endowed with trace-dependent attributes. That is, an object is an observed process: when we look at it we are able to observe the sequence of events in its life, as well as the corresponding sequence of attribute values.

As an illustration, consider a stack of integers as a (passive) object. When we look at it we might observe the following sequences:

events	**attribute values**	
new		empty=true
push(3)	top=3	empty=false
push(7)	top=7	empty=false
pop	top=3	empty=false
pop		empty=true
. . .	. . .	

In a sense, the object stack when observed displays two kinds of behaviour: (1) its traditional trace of atomic operations made upon it (possibly initiated by some other agent, which we ignore here since we are not interested in initiative issues); (2) its corresponding trace of attribute values.

More formally, an object can be defined as a map between behaviours: from the operations behaviour into the attributes behaviour. We adopt the standard terminology within the process community and use the word event instead of "atomic operation". Clearly, events and attributes correspond to "atomic methods" and "slots", respectively, in the terminology of the object-oriented community.

Returning to the stack example, we have to consider the following alphabets of atomic observations (of events and attribute values, respectively):

$$Estack = \{new, pop, drop\} \cup \{push(n): n \in \omega\}$$

$$Vstack = \{(top, n) \mid n \in \omega\} \cup \{(empty, false), (empty, true)\}$$

The former contains all possible events which we may observe in the stack. The latter contains all possible attribute values which we also may observe in that object. (Please note that we previously used the notation "top=n" for the pair (top, n).)

But what are the possible behaviours of the stack? With respect to its traces of events, almost anything is possible: as long as its life starts with the birth operation new, we may subsequently see any sequence of push's and pop's (with the proviso that a pop is not possible when the stack is empty), possibly ending with the death operation drop. With respect to traces of attribute values, we may see sets of pairs (attribute, value) following some rule making them dependent on the observed trace of events.

Actually, as we shall discuss later on, an essential part of an object is precisely this mechanism linking the two observations. We might even argue that this mechanism *is* the object (cf. section 4.1 below).

It is interesting to note how easily we accepted traces of sets of attribute-value pairs for describing the attribute observation behaviour. For instance at a given instant if we observe

$$(top, 7), (empty, false)$$

we say that top = 7 and empty = false. Moreover, if we observe

$$(empty, false)$$

we say that top is undefined and empty = false. Finally, if we observe

$$(top, 7), (top, 9), (empty, false)$$

we would say that top is either 7 or 9 (nondeterminism!) and empty = false.

That is, from the side of the attributes, we naturally adopt a mathematical model supporting both partially defined and nondeterministic attributes: it is enough for that purpose to consider traces of sets of attribute-value pairs.

It is now reasonable to ask if traces of sets of events might also be useful. Indeed, they are: they solve the problem of considering composite objects and their behaviours. As an illustration, consider two isomorphic copies stack1 and stack2 of our original stack. The question is: what is the "joint behaviour" of the composite object stack1 || stack2 ?

We would expect joint traces like the following one:

events	attribute values			
new1		empty1=true		
push1(3)	top1=3	empty1=false		
new2	top1=3	empty1=false		empty2=true
push1(7)	top1=7	empty1=false		empty2=true
pop1	top1=3	empty1=false		empty2=true
pop1 push2(9)		empty1=true	top2=9	empty2=false
pop2		empty1=true		empty2=true
. . .		. . .		

This corresponds to the combination of possible traces of the components, assuming that we accept that two events may happen at the same time: for instance pop1 and push2(9) happen at the same time in the trace above. Thus, we are not restricting ourselves to the pure model of interleaving; although such models of processes are simpler, they are not as powerful as models supporting full concurrency (our model is somewhere in between).

In conclusion, with respect to event behaviour, in order to deal with composition of objects, we also want to consider traces of sets of events. Please note that, as far as processes are concerned, this composition corresponds to parallel composition.

It is useful to introduce here the metaphor of the "blinking observer". Assume that you are an observer who is always blinking (opening and closing your eyes forever). Assume further that you open your eyes for very short periods of time, but that the rate of blinking is as high as needed (you are a very effective observer). Then when you look at an object, you will see its traces of events and of attribute values as follows: Each time you open your eyes you take note of the events happening at that time; and you also take note of the values of the attributes at that time. (This assumes that events always fit into one of your open eyed periods. Naturally, if your rate is not fast enough you may loose some events.)

But assuming that you are a perfect observer, you will see all the traces of all the objects around you. You will notice which events happen at the same time (synchronised) and what are the attribute values at each time. Events of different objects may appear interleaved and/or at the same time.

This metaphor is also useful when understanding object interaction. In general two objects which we want to put together may interact (e.g., by sharing events). As an illustration, consider that stack1 and stack2 above are independent (do not interact) except with respect to creation: they are to be created at the same time. In that case, whenever you observe them when you open your eyes, either new1 and new2 are happening (at the same time) or neither of them is happening. Thus, when two events are shared by two objects, they are always observed together.

The mathematical development of this metaphor is carried out in chapter 3. But it should be noted that already in Go75 a similar view (reviewed in section 2.2) was proposed, but without considering the mechanism for relating event behaviour and attribute-value behaviour. The latter has been under active research in the IS-CORE project (ESS89, ESS90, ES90, SFSE89a+b, SSE87). The two views are brought together in this paper.

It is perhaps useful to take one last insight from the blinking observer metaphor. The observer introduces a fixed time frame which is independent of the "local times" of the observed objects.

As we shall see, this makes life much easier when combining objects. In this respect, the present paper is far away from previous IS-CORE papers which took the position that each object has its own local time structure (namely the structure implied by the trace of events which have already happened).

2.2 Objects as Sheaves

Let's consider the case of an object O which is "transparent" in the sense that it has no hidden events, i.e., all of its behaviour is observable; in the language of software engineering, we could say that none of it is hidden, private, or encapsulated. For such an object, its events *are* its behaviour.

Let us also assume an *ideal observer*, who sees everything that he can, subject to his particular limitations, during his particular lifetime; and let us assume that he leaves behind a *data record* which faithfully records all of his observations, carefully indexed by the time at which they were made. However, it is possible that different observers have different lifetimes, and that observers with the same lifetime observe different things. Let TIME denote the set of all possible lifetimes of ideal observers (later we will give TIME the structure of a category).

For the moment, let's restrict attention to discrete linear time, so that we can assume each observer's lifetime is of the form $\{1,2,\ldots\}$, and that the object he is observing comes into existence at time $t=1$. Thus, an observer sees some "snapshot" $\lambda(t) \in S$ at each moment of time t, where S is the set of all possible instantaneous observations of O, and each observer's data record of a behaviour of O is a (total) function $\lambda: I \longrightarrow S$, where I is some interval of the form $\{1,\ldots,n\}$ and n is the time when he stops watching O. In general, a given object O cannot produce all possible data records $\lambda: I \longrightarrow S$ over an interval I, but only certain "physically realizable" data records. Let $O(I)$ denote the set of all such observable behaviours over I.

Now notice that if J is a subinterval of I, then there is a natural **restriction** function $O(I) \longrightarrow O(J)$ which maps each function $\lambda: I \longrightarrow S$ to the restriction of λ to J, denoted $\lambda \uparrow J: J \longrightarrow S$; for if the snapshots $\lambda(1), \lambda(2), \ldots, \lambda(n)$ can be observed over $I = \{1,\ldots, n\}$ and if $J = \{1,\ldots, n'\}$ with $n' \leq n$, then surely the snapshots $\lambda(1), \lambda(2), \ldots, \lambda(n')$ can be observed over J. If we let $i: J \subset\longrightarrow I$ denote the inclusion, then a reasonable notation for the restriction function is $O(i): O(I) \longrightarrow O(J)$; notice that $O(i)$ goes in the "opposite direction" from i.

All this has a simple categorial formulation, which also suggests the right way to generalize. Namely, let TIME be the subcategory of SET with intervals of the form $\{1,\ldots, n\}$ as objects (including the empty interval, for $n=0$), and with only the inclusions as morphisms. Then O is a contravariant functor from TIME to SET, where $O(i: J \subset\longrightarrow I)$ is the function which restricts functions on I to functions on J.

Clearly, this works just as well if we let TIME be any subcategory of SET with inclusions as morphisms; then $O: \text{TIME}^{op} \longrightarrow$ SET can be any functor such that each $O(I)$ is a set of functions $I \longrightarrow S$, for some fixed set S of snapshots, and such that each $O(i: J \subset\longrightarrow I)$ is a restriction function. As in Go75 and Go90a, let us call such a functor a **pre-object**.

Of course, we can let the snapshots be sets of more primitive observations in order to handle non-determinism, but let us not do so for the moment.

To illustrate, let us consider the stack example again, assuming that everything is visible and deterministic. Then at any moment of time t, an ideal observer will be able to see all of the values on the stack. Thus, the snapshots are finite sequences of natural numbers, i.e., $S = \omega^*$, and each observer's data record has the form of a function $\lambda: \{1, \ldots, n\} \longrightarrow \omega^*$. (Of course, not all such functions are possible, only those consistent with the "life cycle" of a stack; this can be expressed succinctly as: either $\lambda(t) \cap \lambda(t+1) = \lambda(t)$ or $\lambda(t) \cap \lambda(t+1) = \lambda(t+1)$, whenever $0 \leq t \leq n$). Let us denote this object O_S.

Another view of a stack involves observers who see "events" rather than states; their data records are functions $\lambda: \{1, \ldots, n\} \longrightarrow$ Estack*, as in Section 2.1 (but without non-determinism). Let us denote this object O_E.

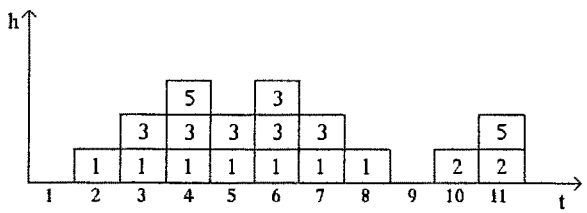
A third view of stacks involves observers who can only see the tops of stacks. Their data records are functions $\lambda: \{1, \ldots, n\} \longrightarrow \{(\text{top}, v) \mid v \in \omega\}^*$. Let us denote this object O_T. (The observation (empty, true) would arise at time t iff $\lambda(t)$ is the empty string.)

What is the relationship among these three objects? It is easy to see that O_E has the most information, and O_T has the least, while O_S lies in between. Thus, there are systematic translations $O_E \xrightarrow{h_1} O_S \xrightarrow{h_2} O_T$ which compute the state from the history, and the top from the state. Following general intuitions about the basic concepts of category theory (Go89), because each object is a functor, we should expect that these translations are *natural transformations.* Indeed, pre-object morphisms *are* natural transformations, and in particular, h_1 and h_2 as well as their composite $h = h_1; h_2$ are natural transformations; the latter is what is called an **object** in this paper. It gives an "interpretation" or "view" of the events in terms of their observable results. On the other hand, O_S is what is usually called a stack in the literature on data types and state machines, while O_E corresponds to the notion of stack studied in the process algebra literature.

We can give a somewhat more exotic version of the data type view of stack, in which the underlying domains include space as well as time. For this purpose, let us define TIME to be the category whose objects are subsets U of $\omega * \omega$ satisfying the following two conditions:

1. $\{ t \mid (t,h) \in U \}$ is an interval of the form $\{1, \ldots, n\}$; let us denote this set t(U); and
2. for each $t \in t(U)$, $\{ h \mid (t,h) \in U \}$ is also an interval of the form $\{1, \ldots, h\}$,

and whose morphisms are inclusions. We let the snapshots be natural numbers. Then a data record is a function of the form $\lambda: U \longrightarrow \omega$ for some U satisfying 1 and 2, as illustrated in the following picture:



in which $U = (\{2\} * [1]) \cup (\{3\} * [2]) \cup (\{4\} * [3]) \cup (\{5\} * [2]) \cup (\{6\} * [3]) \cup (\{7\} * [2]) \cup (\{8\} * [1]) \cup (\{10\} * [1]) \cup (\{11\} * [2])$, where [n] denotes $\{1, \ldots, n\}$.

For the purposes of this paper, only pre-objects in the above sense are needed. But the reader may wonder what all this has to do with sheaves, or indeed, what a sheaf is. Let us assume that TIME is closed under finite union and finite intersection. Then a pre-object O is a **sheaf** iff it satisfies the following condition:

* if $\lambda_1 \in O(U_1)$ and $\lambda_2 \in O(U_2)$, $\lambda_1(t) = \lambda_2(t)$ for all $t \in U_1 \cap U_2$, and $U_1 \cap U_2 \neq \emptyset$, then then there is some $\lambda \in O(U_1 \cup U_2)$ such that $\lambda \upharpoonright U_1 = \lambda_1$ and $\lambda \upharpoonright U_2 = \lambda_2$.

This says that bits of "local" behaviour can be "glued together" if they agree on their overlap, to form larger bits of behaviour. In terms of state machine intuition, this condition says that, relative to the given notion of observation, we have enough information to characterize states (please note that this definition does not presume determinism). In Go75, a pre-object that satisfies this condition is called an **object**, contrary to usage in the present paper.

We conclude this section with some history of the approach it describes. In 1968, Joseph Goguen moved to the University of Chicago to work with Saunders Mac Lane, and began thinking about how to formulate a so-called General Systems Theory in the language of category theory. The basic ideas were that a system is a diagram, its behaviour is its limit, and systems can be interconnected by taking co-limits in the category of systems; see Go71, Go73, GG78. This motivated the approach to specifications in joint work with Rod Burstall on the Clear language and its semantics, which involves taking co-limits in the category of theories (BG77, BG80), and also motivated an examination of the objects that appear in the diagrams representing various kinds of system, which then led to the formulation of objects as sheaves in Go75.

3. Behaviours

This section views an object as an observed process. Thus, an object consists of a process, i.e. of events happening in time, triggering observations which vary in time. According to the object-as-sheaves approach, these are two s-objects (i.e. two objects in the sense of the latter approach) related by an s-object morphism: events happening in time constitute one s-object, observations varying in time constitute another s-object, and "triggering" is expressed as an s-object morphism. In order to avoid confusion, we adopt the term *behaviour* as a synonym for s-object. We also generalize the objects-as-sheaves approach to a purely categorial setting.

We have another terminological problem: the term "object" is used in category theory with quite a different meaning. In order to avoid confusion and stay close to the established terminology, we use the term "c-object" for objects in the categorial sense.

3.1 Atoms and Snapshots

In order to give a uniform treatment to events-in-time and observations-in-time as behaviours, we assume that a universe U of *behaviour atoms* is given. U contains everything atomic for which we might want to say that it may occur at some point in time. Examples are atomic events like create, push(x) for all data elements x, pop and drop as atomic events of a stack object, open, close, credit(m) and debit(m), for all amounts m of money, as atomic events of an

account object, as well as attribute-value pairs like top=0, top=1, empty=false,... as atomic observations for the stack object, or balance=0, overdrawn=false,... as atomic observations for the account object.

Each object will have its own alphabets of event and observation atoms which are subsets of U. We will assume that the subsets of U are the appropriate alphabets of behaviour atoms.

As a basic tool for studying interaction between objects, maps between alphabets of behaviour atoms are needed. This way we can express, say, that an object is embedded in another one (the "environment"), that certain events are shared between different objects, etc.

Assumption 3.1: Let ALPH be a full subcategory of SET such that

(1) its urelements (singleton c-objects whose element has no elements) are the elements of U;

(2) its c-objects include U and all subsets of U;

(3) it is complete and cocomplete (i.e., it has all (small) limits and colimits).

Hereafter, our theoretical developments assume a fixed category ALPH with its "universe" U of urelements. For example, we can either imagine that ALPH has initially been chosen large enough, or that an appropriate "smaller" ALPH has been chosen for that example, to include the necessary atomic behaviours.

Typically, more than one event atom may happen at a given moment in time simultaneously, for example, an entering and a leaving of a nonempty queue. Similarly, we usually do not see single observation atoms at a given moment in time, but rather several of them simultaneously, for example the front element of a queue and its length. Abstracting from events and observations to behaviour atoms, we usually have a *snapshot* $S \subseteq A$ at a given moment in time, where $A \subseteq U$. The power set 2^A is the family of possible snapshots over A; it will be referred to as the *snapshot alphabet* over A.

Behaviour atom alphabets A and B are related by mappings $f: A \longrightarrow B$. A relationship naturally induced between the snapshot alphabets over B and A, respectively, is the (set-valued) inverse mapping $f^{-1}: 2^B \longrightarrow 2^A$. In particular, it expresses the appropriate restriction to a subalphabet in case f is an inclusion, a situation which occurs frequently when dealing with objects and subobjects. For example, if $A \subseteq B$ and if $S \in 2^B$ is a snapshot over B, then $f^{-1}(S) = \{ a \in A \mid f(a) \in S \}$ is the restriction of S to A.

Definition 3.2: Let SNAP denote the category of snapshot alphabets and inverse mappings given by ALPH: its c-objects are the sets 2^A of all subsets of an atom alphabet A, and its morphisms are the inverse mappings $f^{-1}: 2^B \longrightarrow 2^A$ given by $f: A \longrightarrow B$.

There is an obvious functor $F: ALPH^{op} \longrightarrow SNAP$ sending A to 2^A and f to f^{-1}. Clearly, F is an isomorphism of categories, and SNAP is complete as well as cocomplete since ALPH is. As an isomorphism, F preserves limits and colimits. For illustrative purposes as well as for later use, we show how limits in SNAP look, in particular products and equalizers.

Products in SNAP are given by $\prod 2^{A_j} = 2^{\coprod A_j}$ where j ranges over a given index set J, and $\coprod$ denotes disjoint union (coproduct in ALPH). The product morphisms $pr_k: \prod 2^{A_j} \longrightarrow 2^{A_k}$, $k \in J$, are given by $pr_k: \coprod B_j \mapsto B_k$, where $B_j \subseteq A_j$ for $j \in J$, i.e. $pr_k = in_k^{-1}$ where $in_k: B_k \longrightarrow \coprod B_j$ is the injection going with the coproduct. We let $*$ denote the binary (infix) product in SNAP.

Example 3.3: $2^{\{0,1\}} * 2^{\{a,b\}} = 2^{\{0,1,a,b\}}$, and the projections of, say, {0,1,a} are {0,1} and {a}, respectively, i.e. the corresponding restrictions.

As for equalizers in SNAP, let $f,g:A_2 \longrightarrow A_1$ be maps in ALPH, and let $h:A_1 \longrightarrow A_0$ be their coequalizer in ALPH. It is standard to view $\langle f,g \rangle$ as a relation on A_1 (namely $\{ \langle f(a),g(a) \rangle \mid a \in A_2 \}$), and to look at h as representing the equivalence relation generated by $\langle f,g \rangle$ (namely $h(a)=h(b)$ iff a and b are equivalent). By duality and isomorphism, $h^{-1}: 2^{A_0} \longrightarrow 2^{A_1}$ is an equalizer of f^{-1}, g^{-1}: $2^{A_1} \longrightarrow 2^{A_2}$ in SNAP: each subset $C \subseteq A_0$ denotes a set of equivalence classes, and $h^{-1}(C)$ denotes their union $\bigcup C \subseteq A_1$. The unions of equivalence classes obtained this way are precisely those subsets of A_1 which are mapped to the same subset of A_2 by f^{-1} and g^{-1}.

Example 3.4: Let $A_1 = \{0,1,2\}$ and $A_2 = \{a,b,c\}$, and let $f,g:A_2 \longrightarrow A_1$ be given by

$$f : a \mapsto 0 \ , \ b \mapsto 0 \ , \ c \mapsto 1 \ ,$$
$$g : a \mapsto 0 \ , \ b \mapsto 2 \ , \ c \mapsto 1 \ .$$

Then a coequalizer of f and g in ALPH is $h:A_1 \longrightarrow A_0 = \{x,y\}$ given by

$$h : 0 \mapsto x \ , \ 1 \mapsto y \ , \ 2 \mapsto x \ .$$

In SNAP, the equalizer of $f^{-1}, g^{-1}: 2^{\{0,1,2\}} \longrightarrow 2^{\{a,b,c\}}$ is given by $h^{-1}: 2^{\{x,y\}} \longrightarrow 2^{\{0,1,2\}}$ sending $\emptyset$ to $\emptyset$, {x} to {0,2}, {y} to {1}, and {x,y} to {0,1,2}. In fact, these four target sets are precisely those where f^{-1} and g^{-1} coincide: $f^{-1}(\emptyset)=g^{-1}(\emptyset)=\emptyset$, $f^{-1}(\{0,2\})=g^{-1}(\{0,2\})=\{a,b\}$, $f^{-1}(\{1\})=g^{-1}(\{1\})=\{c\}$, and $f^{-1}(\{0,1,2\})=g^{-1}(\{0,1,2\})=\{a,b,c\}$).

3.2 Time Domains and Trajectories

Dynamic behaviour is modelled by attaching behaviour snapshots to points in time. In this section, we discuss suitable models for "points in time" and how they are structured, and how snapshots are "attached" to these points in time.

We note in passing that our approach is in fact more general: we can equally well deal with "points in time-space", i.e. behaviours which do not only extend over time but also - or only - over space. However, the predominant intuition with objects in computing is that they have a temporal but no spatial dimension. So we stick to the usual temporal terminology.

Most generally, our assumption about time is that there are "time domains" which may be related by "morphisms" which are inclusions of one time domain in another.

Definition 3.5: Let TIME be a subcategory of SET with only inclusions as morphisms.

Amazingly enough, we do not need any additional assumptions about the time category. Rather, the restriction to inclusions as morphisms can be dropped without affecting the results presented in this paper. However, we do not have reasonable examples of such general time (-space) structures, and we do not want to strain the reader more than necessary.

Our approach to time covers a wide variety of time models, including discrete and continuous time, linear, branching and partial-order time, as well as finitary and infinitary time. We give two examples of simple and widely used time categories for objects in computing.

Example 3.6: DLF denotes the discrete linear finitary time category. Its c-objects are intervals $[n]=\{1,2,\ldots n\}$ for $n \in \omega$, and its morphisms are $[n] \hookrightarrow [m]$ whenever $n \leq m$. The time domains

are finite intervals, and morphisms reflect prefixing. Naturally, $[0]=\emptyset$.

Example 3.7: DLI denotes the discrete linear infinitary time category. Its c-objects are those of DLF augmented by ω, and its morphisms are those of DLF plus $[n]\subset\!\!\rightarrow\omega$ for each $n\in\omega$. This adds the infinite time domain ω to DLF, having each finite one as a prefix.

Let $S\in$SNAP be a snapshot alphabet, and let TIME be a given category of time domains.

Definition 3.8: A *trajectory over S with respect to* TIME is a map $\lambda:t\longrightarrow S$ for some time domain $t\in$TIME.

A trajectory describes precisely which snapshots occur along the points of its time domain.

Example 3.9: With respect to the DLF time category, a trajectory is a map $\tau:[n]\longrightarrow S$ which corresponds to a finite sequence $<s_1,s_2,\ldots,s_n>$ where $s_i\in S$ for $1\leq i\leq n$, i.e. trajectories are the usual traces. With respect to the DLI time category, we have infinite trajectories $\lambda:\omega\longrightarrow S$ in addition, corresponding to infinite sequences $<s_1,s_2,\ldots>$ where $s_i\in S$ for $i\in\omega$.

Please note that our notion of trajectory generalizes the notion of trace in three respects: we have generalized time domains, we have snapshots (sets of atoms) at each point in time, and we abstract from what occurs along time domains: events or observations - or something else.

Motivated by the event case, we say that a trajectory $\lambda:t\longrightarrow S$ "makes a pause" at point $p\in t$ iff $\lambda(p)=\emptyset$, i.e. nothing happens at point p in time domain t.

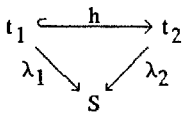
Trajectories over the same snapshot alphabet S are naturally related via TIME morphisms, giving rise to a category of trajectories over S and TIME.

Definition 3.10: Let $\lambda_1:t_1\longrightarrow S$ and $\lambda_2:t_2\longrightarrow S$ be trajectories. A *trajectory morphism* $h:\lambda_1\longrightarrow\lambda_2$ is a TIME morphism $h:t_1\longrightarrow t_2$ such that $\lambda_1=h;\lambda_2$.

TRJ(TIME,S) denotes the category of trajectories over S with respect to TIME, with trajectory morphisms as defined above. We will also write TRJ(S) or simply TRJ if the rest is clear from context.

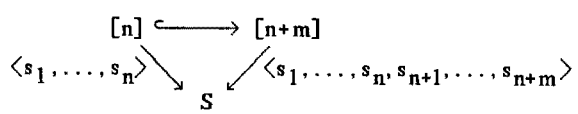
The construction of TRJ(TIME,S) from TIME and S is an instance of the well known "comma category" construction (cf. GB84).

The situation is depicted by the following commutative diagram.



Notice that this means that λ_1 is the *restriction* of λ_2 to the subdomain t_1 of t_2. We will also write $\lambda_1\leq\lambda_2$ iff there is a morphism from λ_1 to λ_2 (there is at most one).

Example 3.11: In the DLF and DLI time models, trajectories are maps $[n]\longrightarrow S$ (or $\omega\longrightarrow S$) which correspond to finite (or infinite) sequences $\langle s_1,s_2,\ldots\rangle$. Trajectory morphisms correspond to prefixes:



3.3 Category of Behaviours

A behaviour is defined as a set of trajectories. Intuitively, a behaviour displays the possible life cycles (with respect to events or observations) an object can go through. In what follows, we assume that the TIME category is fixed once and for all.

Definition 3.12: A *behaviour over* S is a subcategory inclusion

$$(S,\Lambda): \Lambda \hookrightarrow \mathrm{TRJ}(S)$$

where S is a snapshot alphabet in SNAP, with the property that the constant map $\emptyset_t: t \longrightarrow \{\emptyset\}$ is in Λ for each $t \in \mathrm{TIME}$.

The latter condition says that the "empty trajectory" (permanent pause) over any time domain is always possible. This is needed later for technical reasons, but it also has an intuitive appeal in its own right. Moreover, we will most often assume in examples that behaviours are "closed with respect to pauses", i.e. if λ_1 is in Λ and λ_2 can be obtained from λ_1 by inserting and omitting pauses, then also λ_2 is in Λ. This is a natural condition in cases where we deal with "asynchronous" behaviour, i.e., where only the relative ordering of nonempty snapshots in time matters, not the absolute time points when they occur. Please note that λ_1 and λ_2 as defined above, i.e. being "the same modulo pauses", will in general be trajectories over different time domains.

Definition 3.13: Let (S_1,Λ_1) and (S_2,Λ_2) be behaviours. A *behaviour morphism* is a functor

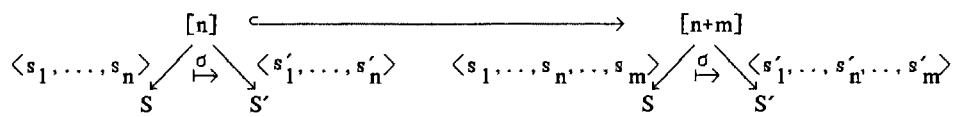
$$\sigma : (S_1,\Lambda_1) \longrightarrow (S_2,\Lambda_2)$$

such that $\mathrm{dom}\,\lambda = \mathrm{dom}\,\sigma(\lambda)$ for each trajectory $\lambda \in \Lambda_1$. Moreover, if $\emptyset_t: t \longrightarrow \{\emptyset\}$ is the "permanent pause" trajectory over t, then $\sigma(\emptyset_t) = \emptyset_t$.

That is, λ and $\sigma(\lambda)$ always have the same underlying time domain (they have "the same length"), and permanent pauses are always sent to permanent pauses.

This is rather general and might look strange, one would perhaps expect a SNAP morphism between S_1 and S_2 as part of a behaviour morphism. But the generality is needed. For example, stack event behaviours and corresponding stack observation behaviours should be related by a behaviour morphism. While it is very well possible to associate the observation top=k with a push(k) event at any point in time, there is no single observation which can be associated with a pop event: any top value is possible, depending on context. Interesting special cases of behaviour morphisms, however, do go with an underlying SNAP morphism, cf. definition 3.17 below.

Example 3.14: In the DLF and DLI time categories, we have the following situation.



This means that σ is "monotonic": prefixes are sent to prefixes. Thus, σ acts as a "state function" where the snapshot at a given position depends on the "past" (snapshots at previous positions) only. These time models thus exclude "prophecy" effects (which can, however, be achieved with other time models, for instance the discrete versions of DLF and DLI obtained by omitting all morphisms).

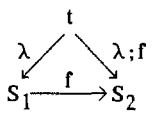
It is an easy exercise to verify that behaviours and behaviour morphisms as defined above do form a category.

Definition 3.15: Given TIME and SNAP, the category BHV(TIME,SNAP) is the category of all behaviours over some snapshot alphabet $S \in SNAP$ with respect to TIME, and all behaviour morphisms among them, as defined above. We will also write BHV(SNAP) or simply BHV if the rest is clear from context.

Behaviours (S_1, Λ) and $(S_1 \cup S_2, \Lambda)$, with the same Λ, are isomorphic in BHV, i.e. behaviours do not change essentially if we enlarge or restrict the underlying snapshot alphabet, as long as all snapshots occurring in Λ are present. Therefore, we sometimes write just Λ instead of (S,Λ), meaning that S is understood to be the set of all snapshots occurring in trajectories in Λ.

Remark 3.16: With the general approach presented here, there is no problem to handle "transactions", i.e. elements $\lambda : t \longrightarrow S$ of a behaviour which are given a status of "atomicity" by including them into the set A of behaviour atoms underlying S (i.e. $S = 2^A$). This way, transactions can be nested arbitrarily. An example of transactions is given in section 4.3.

Definition 3.17: A behaviour morphism $\sigma : (S_1, \Lambda_1) \longrightarrow (S_2, \Lambda_2)$ is called *oblivious* iff it is of the form $\sigma(\lambda) = \lambda ; f$ for some fixed SNAP morphism $f : S_1 \longrightarrow S_2$.



If σ is oblivious, then, at each point in time, $\sigma(\lambda)$ depends on λ at the same point in time only, not on any λ components "before" or "after" or "concurrently". Assuming the DLF or DLI time model, σ is oblivious iff $\sigma(\tau\lambda) = \sigma(\tau)\sigma(\lambda)$ holds for any finite sequence τ and any sequence λ; hence, $\sigma(s_1 s_2 \ldots) = \sigma(s_1)\sigma(s_2) \ldots$.

Theorem 3.18: Given categories SNAP of behaviour snapshots and TIME of time domains, the category BHV(SNAP,TIME) of behaviours over SNAP wit respect to TIME is complete.

Proof: We show that BHV has products and equalizers.

As for products, let (S_j, Λ_j), $j \in J$, be a family of behaviours. Let $pr_j : S \longrightarrow S_j$, $j \in J$, be the product in SNAP (cf. section 3.1) where $S = \prod S_j$. Let $\Lambda \subset\!\!\!\rightarrow TRJ(S)$ be the full subcategory consisting of all trajectories $\lambda \in TRJ(S)$ such that $\lambda ; pr_j \in \Lambda_j$ for each $j \in J$. Let $\pi_j : \Lambda \longrightarrow \Lambda_j$, $j \in J$, be the oblivious behaviour morphism given by $\pi_j(\lambda) = \lambda ; pr_j$ for each $j \in J$. Then the π_j, $j \in J$, constitute a product in BHV.

In order to verify this, let $\sigma_j : (S', \Lambda') \longrightarrow (S_j, \Lambda_j)$, $j \in J$, be a family of behaviour morphisms (not necessarily oblivious!). Then each trajectory $\lambda' \in \Lambda'$ is sent to a trajectory $\lambda_j = \sigma_j(\lambda') \in \Lambda_j$ for each $j \in J$. Let λ be the trajectory $\lambda \in \Lambda$ defined by $\lambda_j = \pi_j(\lambda)$ for each $j \in J$ (it is clear that there is exactly one $\lambda \in \Lambda$ satisfying this condition). Let $\sigma : \Lambda' \longrightarrow \Lambda$ be the map defined this way. σ preserves the time structure so that it is a functor: if $\lambda'_1 \leq \lambda'_2$, then $\sigma_j(\lambda'_1) \leq \sigma_j(\lambda'_2)$ for each $j \in J$, from which we conclude by construction that $\sigma(\lambda'_1) \leq \sigma(\lambda'_2)$. Clearly, $\sigma_j = \sigma ; \pi_j$ for each $j \in J$, and σ is the only map satisfying this equation. Moreover, $dom\,\lambda' = dom\,\sigma_j(\lambda') = dom\,\lambda_j = dom\,\lambda$ for each $j \in J$, and $\sigma(\emptyset_t) = \emptyset_t$ for each time domain t.

Thus, σ is a behaviour morphism, and it is the only one from (S', Λ') to (S, Λ) satisfying $\sigma ; \pi_j = \sigma_j$ for all $j \in J$. This verifies that the π_j, $j \in J$, constitute a product in BHV.

Essentially, the product is taken componentwise along time domains.

As for equalizers, let $\sigma,\rho:(S_1,\Lambda_1)\longrightarrow(S_2,\Lambda_2)$ be two behaviour morphisms. An equalizer of σ and ρ is constructed as in SET: it is given by the inclusion $\xi:(S_0,\Lambda_0)\ ^{c}\!\!\longrightarrow(S_1,\Lambda_1)$ where S_0 is some subset of S_1 containing all snapshots occuring in Λ_0 (different choices lead to isomorphic behaviours), and $\Lambda_0=\{\lambda\in\Lambda_1 \mid \sigma(\lambda)=\rho(\lambda)\}$.

In order to verify this, let $\tau':(S',\Lambda')\longrightarrow(S_1,\Lambda_1)$ be a behaviour morphism satisfying $\tau';\sigma=\tau';\rho$. By construction, $\tau'(\Lambda')\subseteq\Lambda_0$ so that there is exactly one map $\tau:\Lambda'\longrightarrow\Lambda_0$ satisfying $\tau;\xi=\tau'$. Since τ essentially *is* τ' (with the range restricted to Λ_0), it is obvious that τ is a behaviour morphism, and it is the only one from (S',Λ') to (S_1,Λ_1) satisfying $\tau;\xi=\tau'$. This verifies that ξ constitutes an equalizer of σ and ρ. □

3.4 Parallel Composition

Limits in a behaviour category reflect parallel composition of behaviours. From the proof of theorem 3.17, we see that products are constructed "pointwise" along a common time domain by taking the disjoint unions of snapshots.

Example 3.19: Assuming the DLF time category, let Λ_1 and Λ_2 be given as follows.

$$\Lambda_1 = \{\ \langle\{2,4\},\{5\}\ ,\ \{1\}\ \rangle\ ,\ \langle\ \{4\}\ ,\ \{1,3\}\ \rangle\}^{\#}$$

$$\Lambda_2 = \{\ \langle\ \{a\}\ ,\ \{d\},\{b,c\}\rangle\ ,\ \langle\{a,c\},\{b,d\}\rangle\}^{\#}$$

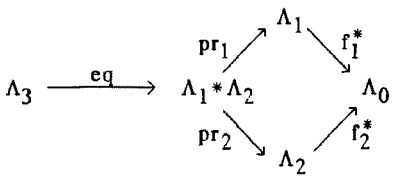
where $^{\#}$ denotes closure with respect to pauses. That is, all trajectories which can be obtained from the two in Λ_1 shown above, by inserting pauses, are also in Λ_1, and correspondingly for Λ_2.

Let $\Lambda_0 = \{\ \langle\{x\},\{y\}\rangle\ \}^{\#}$.

Let $f_1 : x\mapsto 4\ ,\ y\mapsto 1$

and $f_2 : x\mapsto a\ ,\ y\mapsto d$,

indicating that we want to synchronize on 4≡a and 1≡d. Let $f_1^*:\Lambda_1\longrightarrow\Lambda_0$ and $f_2^*:\Lambda_2\longrightarrow\Lambda_0$ be those oblivious behaviour morphisms obtained by applying f_1^{-1} or f_2^{-1}, respectively, to each point in time along each trajectory. Then a pullback object Λ_3 of f_1^* and f_2^* is an equalizer object of $pr_1;f_1^*$ and $pr_2;f_2^*$, as shown in the following diagram.



The pullback object Λ_3 is given by all "interleavings" of behaviours in Λ_1 and Λ_2, appropriately synchronized, appearing as componentwise "union" of behaviours interspersed with $\emptyset$ (and made equal in length this way):

$\langle\{x,2\}\ ,\ \{5\}\ ,\ \{y\}\ ,\ \{b,c\}\rangle$	from	$\langle\{2,4\},\{5\},\{1\},\emptyset\rangle$	and	$\langle\{a\},\emptyset,\{d\},\{b,c\}\rangle$,
$\langle\{x,2,c\}\ ,\ \{5\}\ ,\ \{y,b\}\rangle$	from	$\langle\{2,4\},\{5\},\{1\}\rangle$	and	$\langle\{a,c\},\emptyset,\{b,d\}\rangle$,
$\langle\{x\}\ ,\ \{y,3\}\ ,\ \{b,c\}\rangle$	from	$\langle\{4\},\{1,3\},\emptyset\rangle$	and	$\langle\{a\},\{d\},\{b,c\}\rangle$,
$\langle\{x,c\}\ ,\ \{y,3,b\}\rangle$	from	$\langle\{4\},\{1,3\}\rangle$	and	$\langle\{a,c\},\{b,d\}\rangle$,

plus all behaviours obtainable from these by inserting any number of pauses in any place.

Another (isomorphic) choice of the pullback object is obtained if we keep a and 4 distinct instead of merging them into a single symbol x, and the same with d, 1 and y, respectively. The "identifications" x≡a≡4 and y≡d≡1 are then reflected by the fact that a and 4 (or d and 1, respectively) always appear together in a synchronization set: both are in the set, or neither of them is.

4. Objects

Objects are defined as behaviour morphisms, capturing the idea of "processes endowed with observations". Object morphisms are pairs of behaviour morphisms between the process and observation parts, respectively, satisfying a natural compatibility condition. The category of objects established this way is shown to be cocomplete. This means, for instance, that aggregation of objects is compositional. Various forms of object inheritance can be expressed as object morphisms, and also object reification (sometimes called refinement) can be expressed this way.

4.1 Category of Objects

Let BHV be a complete behaviour category as described in the previous sections. Intuitively speaking, an object tells how observations in time (and/or space) depend on events in time (and/or space). This is appropriately modelled by a morphism in BHV.

Definition 4.1: An *object* is a behaviour morphism $ob: (E,\Lambda) \longrightarrow (V,\Omega)$ in BHV.

Intuitively, the first behaviour is the "active" part (process), and the second behaviour is the "passive" part (observation). That is why we use E and V for the respective snapshot alphabets. ob describes how the process "triggers" its observations.

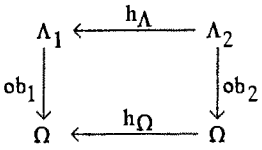
We write $ob: \Lambda \longrightarrow \Omega$ if E and V are clear from context. For illustration, we refer to the examples in section 2.1.

There are two obvious ways to derive new objects from given ones, namely by "triggering" and by "observing" via respective behaviour morphisms, as shown in the following diagrams.

$$\begin{array}{ccc} \Lambda_1 & \xleftarrow{h_\Lambda} & \Lambda_2 \\ {\scriptstyle ob_1}\downarrow & & \\ \Omega_1 & & \end{array} \qquad\qquad \begin{array}{ccc} & & \Lambda_2 \\ & & \downarrow{\scriptstyle ob_2} \\ \Omega_1 & \xleftarrow{h_\Omega} & \Omega_2 \end{array}$$

On the left hand side, object ob_1 is "triggered" via h_Λ in the sense that the composed object $h_\Lambda;ob_1$ has Λ_2 as its process part and h_Λ tells ob_1 how to "obey the commands" in Λ_2. Analogously, on the right hand side, ob_2 is "observed via" h_Ω in the sense that h_Ω tells how to "interpret" the observations of ob_2 in terms of behaviour Ω_1. In the special case where h_Λ and h_Ω are restrictions on snapshot alphabets going with inclusions on the respective atom alphabets, "triggering" means disregarding the events in Λ_2 which are not in the scope of Λ_1, and "interpreting" observations means viewing only those in the scope of Ω_1 (cf. Definition 4.5 below).

An object morphism is a relationship between objects ob_1 and ob_2 where the process part of ob_2 triggers ob_1 via some behaviour morphism h_Λ while, at the same time, the observation part of ob_1 observes ob_2 via some other behaviour morphism h_Ω, in such a way that "ob_1 triggered via h_Λ" is the same object as "ob_2 observed via h_Ω", i.e. the following diagram commutes.



Definition 4.2: Let $ob_i: \Lambda_i \longrightarrow \Omega_i$, i=1,2, be objects. An *object morphism* $h: ob_1 \longrightarrow ob_2$ is a pair $(h_\Lambda: \Lambda_2 \longrightarrow \Lambda_1 , h_\Omega: \Omega_2 \longrightarrow \Omega_1)$ of behaviour morphisms such that $h_\Lambda; ob_1 = ob_2; h_\Omega$ holds.

In the next subsection, we will explore special cases of object morphisms which model different kinds of object inheritance. The "oblivious" object morphisms (cf. Definition 3.17) to be defined next are a sort of standard case. They play an essential role for studying inheritance.

Definition 4.3: An object morphism $h: ob_1 \longrightarrow ob_2$ is called *oblivious* iff both h_Λ and h_Ω are oblivious behaviour morphisms.

A word is in order about the choice of direction for object morphisms. Basically, this is a matter of taste and we could have defined them the other way around. Our choice is motivated by the direction of maps on the underlying atom alphabets in the case of oblivious morphisms. If $h: ob_1 \longrightarrow ob_2$ is such an object morphism, i.e., if h_Λ and h_Ω are oblivious, then $h_\Lambda: \Lambda_2 \longrightarrow \Lambda_1$ comes from a map $g_\Lambda^{-1}: S_2 \longrightarrow S_1$ on snapshot alphabets which in turn comes from a map $g_\Lambda: A_1 \longrightarrow A_2$ between the underlying atom alphabets ($S_i = 2^{A_i}$ for i=1,2). The same holds for the Ω part. If g_Λ and g_Ω are inclusions, then the corresponding object morphism goes from the "part" to the "whole", describing the embedding of an object into an environment (which is an object, too). The argument that the arrows should go the other way is almost as compelling: this is the way that the arrows actually go in the diagram above, and also, it leads to using limits to compute the behaviours of systems, in accord with the general "dogmas" of Go89.

Of course, objects and object morphisms form a category. We denote this category by OB.

Theorem 4.4: OB is cocomplete.

Proof: Taking morphisms of a given category K as c-objects and commutative squares in K as morphisms of a new category L is a well known categorial construction. We use the notation L = Mor(K). L can be described as a comma category (cf. GB84), namely L = (K/K) (we identify K with the identity functor on it). The category of objects is $OB = Mor(BHV)^{op} = (BHV/BHV)^{op}$. Since BHV is complete (Theorem 3.18), we conclude from well-known theorems of category theory that Mor(BHV) is complete; thus, OB is cocomplete. □

From the example in section 3.4, it is clear that colimits in OB can be utilized to model (parallel) composition of objects. This theorem, therefore, gives a very general basis for compositional semantics of object-oriented systems: we describe single objects as behaviour morphisms, interaction between objects by object morphisms, and we obtain the community of all interacting objects as one composite object, the colimit object in OB, with the universal cocone describing how the single objects are embedded in (or "part of") the community.

The above theorem holds for arbitrary time (and/or space) categories TIME, covering quite a variety of process models going far beyond mere interleaving. Please remember that "transactions" are also included (cf. Remark 3.16): we may very well decide to pick an element of a behaviour over a set A of atomic events and put it into A so that it plays the role of an atomic event. Transactions can be "synchronized" with other atomic events (thus with other transactions as well) by putting them into the same snapshot. Please remember that, at each time instant of a transaction, we may again have transactions within the snapshot, etc., i.e. transactions can be nested.

From the dual of the object category, OB^{op}, into the behavior category BHV, we have two obvious forgetful functors, giving the underlying event and observation behaviours, respectively:

$$\underline{\Lambda} : OB^{op} \longrightarrow BHV$$

"forgets" the observation parts: it sends each object $ob : \Lambda \longrightarrow \Omega$ to its event behaviour Λ, and each object morphism $h : ob_1 \longrightarrow ob_2 = (h_\Lambda : \Lambda_2 \longrightarrow \Lambda_1,\ h_\Omega : \Omega_2 \longrightarrow \Omega_1)$ to its event behaviour morphism $h_\Lambda : \Lambda_2 \longrightarrow \Lambda_1$. Similarly,

$$\underline{\Omega} : OB^{op} \longrightarrow BHV$$

"forgets" the event parts: it sends $ob : \Lambda \longrightarrow \Omega$ to Ω, and $h : ob_1 \longrightarrow ob_2$ to $h_\Omega : \Omega_2 \longrightarrow \Omega_1$.

According to definitions 4.1 and 4.2, each object ob *is* the behaviour morphism

$$ob : \underline{\Lambda}(ob) \longrightarrow \underline{\Omega}(ob) ,$$

and each object morphism $h : ob_1 \longrightarrow ob_2$ *is* the commutative diagram

$$\begin{array}{ccc} \underline{\Lambda}(ob_1) & \xleftarrow{\underline{\Lambda}(h)} & \underline{\Lambda}(ob_2) \\ {\scriptstyle ob_1}\downarrow & & \downarrow{\scriptstyle ob_2} \\ \underline{\Omega}(ob_1) & \xleftarrow{\underline{\Omega}(h)} & \underline{\Omega}(ob_2) \end{array}$$

This means that the category OB of objects can be described as the dual of a comma category,

$$OB = (BHV/BHV)^{op} ,$$

where the category BHV is identified with the identity functor on it (cf. proof of theorem 4.4), and $\underline{\Lambda}$ and $\underline{\Omega}$ are the two projection functors. The data given above can also be interpreted as describing a *natural transformation*

$$\underline{ob} : \underline{\Lambda} \Longrightarrow \underline{\Omega} .$$

The following diagram shows how OB, BHV, $\underline{\Lambda}$, $\underline{\Omega}$ and $\underline{ob}$ are related:

$$\begin{array}{ccccc} & & OB^{op} & & \\ & \swarrow & \Downarrow \underline{ob} & \searrow & \\ \underline{\Lambda} & & & & \underline{\Omega} \\ & \searrow & & \swarrow & \\ & & BHV & & \end{array}$$

From general results in category theory, we conclude that $\underline{\Lambda}$ and $\underline{\Omega}$ are cocontinuous.

4.2 Object Inheritance

In this section, we study some aspects of inheritance in object-oriented approaches. We show that object morphisms can be used to formalize several kinds of inheritance as relationships between objects. We avoid, however, the word "inheritance" as a technical term because of the notorious confusion surrounding it. In particular, we discuss strict inclusion, weak inclusion and enclosure.

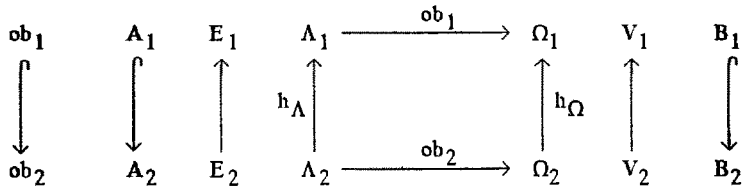
4.2.1 Strict Inclusion

In an intuitive sense, an inclusion morphism $h: ob_1 \hookrightarrow ob_2$ as defined below describes how the "part" ob_1 is embedded in an "environment" or "complex object" ob_2 such that ob_1 is "encapsulated within" ob_2 in the sense that no events outside ob_1 can affect observations within ob_1. Typical examples are *engine* $\hookrightarrow$ *car*, *memory* $\hookrightarrow$ *computer*, etc.

A specific application for this kind of object morphism is "object sharing", i.e. the inclusion of one object into several other objects. The case that just single events are shared has been utilized by the first and third authors as a means for synchronous and symmetric communication between objects (ESS90,ES90, SFSE89a,SSE87).

Definition 4.5: Let $h: ob_1 \longrightarrow ob_2$ be an oblivious object morphism. If the underlying maps on atomic events $g_\Lambda: A_1 \hookrightarrow A_2$ and atomic observations $g_\Omega: B_1 \hookrightarrow B_2$ are inclusions, we call h an *inclusion morphism* and write $h: ob_1 \hookrightarrow ob_2$.

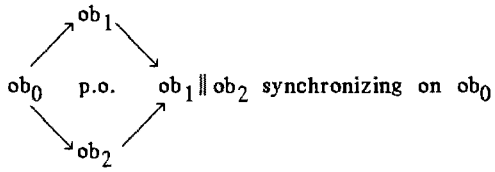
Please note that, for an inclusion morphism h, h_Λ and h_Ω are not inclusions themselves but restrictions on snapshot alphabets resulting from inclusions on the underlying atom alphabets. The following diagram illustrates the situation.



As for inheritance, the inclusion morphism says that ob_2 "inherits" the atomic events and observations from ob_1 such that ob_1 observations are "views" of ob_2 observations, and ob_2 event behaviours (life cycles) are "enrichments" of ob_1 life cycles. The morphism condition says that any permissible enrichment of an ob_1 life cycle $\lambda_1 \in \Lambda_1$, when observed in ob_2, gives rise to the same observation in the view of ob_1.

For example, restricting a computer life cycle to memory events and observing the latter gives the same as observing the entire computer life cycle and restricting attention to memory observations only. That is, only memory events can influence memory observations, there is no way that non-memory computer events can have an effect on the observable behavior of the memory. In this sense, the memory is an "encapsulated object within" the computer. This does not mean that no communication is possible: the computer can "use" memory events - but only these - to operate on its memory.

The standard application of inclusion morphisms is to describe the composition of complex objects, i.e. the aggregation of objects, sharing encapsulated subobjects: if $in_i: ob_0 \longrightarrow ob_i$, $i \in \{1,2, \ldots, n\}$, then the aggregation $ob_1 \| ob_2 \| \ldots \| ob_n$ synchronizing on ob_0 is the colimit of the diagram consisting of the in_i's, and the universal cocone describes how each ob_i is embedded into the aggregation. The following diagram illustrates this for the case n=2, where the colimit is a pushout.

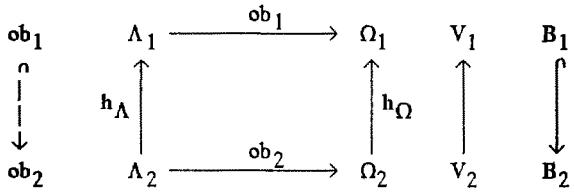


4.2.2 Weak Inclusion

If we keep the inclusion idea just for the observation part and liberalize the event part to arbitrary behaviour morphisms, we arrive at the concept of "observation inclusion morphism" which models a weaker form of inheritance: environment events can affect local observations directly.

Definition 4.6: Let $h: ob_1 \longrightarrow ob_2$ be an object morphism. If the constituent observation behaviour morphism h_Ω is oblivious and its underlying map on atomic events $h_\Omega: B_1 \overset{c}{\longrightarrow} B_2$ is an inclusion, we call h an *observation inclusion morphism* and write $h: ob_1 \overset{c}{\dashrightarrow} ob_2$.

The following diagram may help to understand this situation.



As in the case of inclusion morphisms, ob_1 observations are "views" of ob_2 observations, but ob_2 event life cycles may "trigger" ob_1 in an arbitrary way. This can be utilized to model "loose" embeddings of a "part" ob_1 into an "environment" or "complex object" ob_2 where environment events can affect local observations, but only in a way which can be simulated by some local events: h_Λ tells how the local effect of global life cycles is simulated locally.

In the linear discrete time models, this simulation of global life cycles by local ones must preserve prefixes. Consequently, in cases where the life cycle set is prefix-closed, this simulation can only happen in an event-by-event way: global events "call" local events (cf. SE90, SEC90). In order to illustrate this, consider a user using a stack (which may be shared by other users). We model this by including the stack object weakly into the user object:

$$\text{stack} \overset{c}{\dashrightarrow} \text{user} .$$

For each stack event pop, push(k), etc., we assume that the user has a corresponding private event call-pop, call-push(k), etc., "calling" the corresponding stack event in the above sense. We assume that a user uses only his private calls for operating on the stack, not the stack events directly. The weak inclusion stack $\overset{c}{\dashrightarrow}$ user is then established by replacing stack calls

by the corresponding stack events and forgetting about all other user events, mapping each user life cycle to a stack life cycle this way.

The point in this construction is that arbitrarily many users can be hooked to the same stack this way, sharing it weakly in the sense that everybody can operate on it, and everybody can observe all the effects, also those caused by others. Each user may, however, define additional attributes which are changed when he calls the stack (for local bookkeeping or so), and which cannot be observed, let alone changed, by the others.

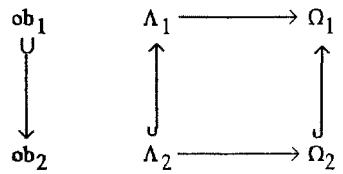
The aggregate object - users sharing a stack - is obtained as a colimit in OB, in much the same way as in the strict inclusion case.

4.2.3 Enclosure

The enclosure morphisms to be defined next are much more liberal than strict and weak inclusions: if ob_1 encloses ob_2, then event as well as observation behaviours of ob_2 are included in those of ob_1, respectively (please note that the inclusions are on behaviours and go in the opposite direction), such that ob_2 in isolation works in exactly the same way as it works in the context of ob_1 - as long as ob_1 "doesn't interfere".

Definition 4.7: Let $ob_i: \Lambda_i \longrightarrow \Omega_i$, $i=1,2$, be objects, and let $\Lambda_2 \subseteq \Lambda_1$ and $\Omega_2 \subseteq \Omega_1$ such that the inclusions form an object morphism $h: ob_1 \longrightarrow ob_2$. In this case, we call h an *enclosure morphism* and use the notation $h: ob_1 \supset\!\!\longrightarrow ob_2$.

An enclosure morphism is illustrated by the following diagram.



In general, enclosure morphisms are not oblivious, that is why we do not show the underlying alphabets in the diagram. All that can be said about the snapshot alphabets is that each snapshot occuring in Λ_2 must also occur in Λ_1, and the same for Ω_2 and Ω_1. Please note that this does not necessarily mean inclusion between the respective snapshot alphabets.

Intuitively, if ob_1 encloses ob_2, ob_1 acts exactly like ob_2 as long as only ob_2 events occur. However, once a "new" ob_1 event happens, nothing is incurred for ob_1 any more (not even in retrospect, operationally speaking). We have been experimenting with additional conditions ensuring that "old" events maintain their effects on observations also in case "new" events occur (see also Gu90). This subject deserves further study.

Enclosure morphisms seem to have their methodological virtue in modelling "roles" of objects, in the sense in which patient, employee, car driver, tax payer, etc. are roles of person: patient $\supset\!\!\longrightarrow$ person, employee $\supset\!\!\longrightarrow$ person, car-driver $\supset\!\!\longrightarrow$ person, tax-payer $\supset\!\!\longrightarrow$ person, etc. In fact, patients, employees, car drivers, tax payers, etc., should basically behave like persons.

Please note that this situation is very different from aggregation forming complex objects. A person is not the complex object with parts patient, employee, etc., and the patient, employee, etc. objects are not aggregations sharing the person object as a common part either. Indeed, the

latter would mean that patient, employee, etc. life cycles are proceeding concurrently all the time. Rather, a person's behaviour shows phases where, say, she is a patient, other phases where she is an employee, and still other phases where she is both at the same time. The mathematics of enclosure reflects this appropriately: when taking colimits, several roles of the same object have the latter as colimit object and the enclosures as universal cocone, so nothing new is constructed.

A thorough treatment of all aspects of inheritance is outside the scope of this paper. Besides inheritance between objects as discussed here, inheritance between object *types* and object *classes* (which we do not treat in this paper) have to be taken into account, as well as inheritance between *specifications* of objects, object types, and object classes (see also HC89). This area requires further study.

4.3 Object Reification

Our approach is based on reification as an implementation relationship between objects as studied in ES90. Intuitively, reification describes the relationship between an "abstract interface" implemented on top of a "base interface". That is, we deal with reification as a relationship between objects. This has to be distinguished from the relationship between specifications of "describing in more detail", and also from the relationship between a specification and an object which "complies with" the specification. Both are sometimes called "implementation", too.

Reification has been studied extensively in the field of abstract data types and their specification, starting with the pioneering paper GTW78. Essential ideas can already be found in Ho72. The following example is taken from ES90. It is treated in Go90b from a specification point of view.

Example 4.8: Let the stack object in section 2.1 be given, with the following behaviour atoms.

stack:	event atoms	new, drop, push(i) for $i \in$ int, pop
	observation atoms	top=i for $i \in$ int, empty?=b for $b \in$ bool

We want to implement this "abstract" stack object on top of an array with a top pointer nvar (which is a variable over natural numbers), having the following behaviour atoms (cf example 4.1 in ES90).

array:	event atoms	create, destroy, set(n,i) for $n \in$ nat and $i \in$ int
	observation atoms	conts(n)=i for $n \in$ nat and $i \in$ int
nvar:	event atoms	open, close, asg(n) for $n \in$ nat
	observation atoms	val=n for $n \in$ nat

The intuitive meaning of these atoms should be clear. Intuitively, an implementation of stack over array and nvar would do the following two things:

(1) *encode* each stack event by a "transaction" over the base, i.e. a sequence of array and nvar events, for instance

new $\mapsto$ <create;open;asg(0)>
drop $\mapsto$ <close;destroy>
push(i) $\mapsto$ <set([val],i);asg([val]+1)>
pop $\mapsto$ <asg([val]-1)>

Here, [val] denotes the current value of the attribute val of nvar.

(2) *decode* each observation over the base attributes as an observation over the stack attributes, for instance

top = [conts([val]-1)]
empty? = equal?([val],0)

Since events from several base objects are interleaved in the above encoding, we should look at the composite object bas = array‖nvar as being the base, rather than some collection of base objects. Thus, we may assume that the base is just a single object.

Please note that the base "transaction" by which a stack event is encoded will lead to different base traces for the same stack event, depending on context. For instance, pop can mean <asg(0)> or <asg(1)> or . . . , and push(1) can mean <set(0,1);asg(1)> or <set(1,1);asg(2)> or . . . , depending on the value of val in the state where pop or push(1) occurs, respectively.

Each stack life cycle, for instance

$$<\text{new} ; \text{push}(1) ; \text{push}(2) ; \text{pop} ; \text{push}(1) ; \text{pop}^{*} ; \text{pop} ; \text{drop}> ,$$

can be transformed into a sequence of base transactions by means of the above encoding:

$$<<\text{create} ; \text{open} ; \text{asg}(0)> ; <\text{set}(0,1) ; \text{asg}(1)> ; <\text{set}(1,2) ; \text{asg}(2)> ; <\text{asg}(1)> ; <\text{set}(1,1) ; \text{asg}(2)> ; <\text{asg}(1)>^{*} ; <\text{asg}(0)> ; <\text{close} ; \text{destroy} >> .$$

Please note that the two sequences shown above have the same length, if we count each transaction in the second sequence as just one atomic step. The asterisk marks the same position in both sequences. If we "unfold" the latter transaction sequence into a flat sequence of base events, we arrive at the following sequence. Please note that it is longer than the above two sequences.

$$<\text{create} ; \text{open} ; \text{asg}(0) ; \text{set}(0,1) ; \text{asg}(1) ; \text{set}(1,2) ; \text{asg}(2) ; \text{asg}(1) ; \text{set}(1,1) ; \text{asg}(2) ; \text{asg}(1)^{*} ; \text{asg}(0) ; \text{close} ; \text{destroy} > .$$

Here, the asterisk marks a "corresponding" position, not the same one, because the underlying time domains are different. Encoding amounts to "compiling" stack life cycles into life cycles over base transactions of the same length, and the latter are obtained by "folding" base life cycles into transactions.

The result of compiling a stack life cycle should be "executable", i.e. its unfolded version should be a valid base life cycle, and the observations along the corresponding folded version, when decoded as stack observations, should comply with the given stack behaviour. For instance, at the end of the initial trace of the above base life cycle ending at *, we have the following observation snapshot:

val = 1 conts(0) = 1 conts(1) = 1

This base observation snapshot decodes as the following stack observation snapshot:

top = [conts([val]-1)] = [conts(0)] = 1
empty? = equal?([val],0) = equal?(1,0) = false

This is the correct observation snapshot at the end of the corresponding stack trace, i.e. the initial trace of the above stack life cycle ending at * . □

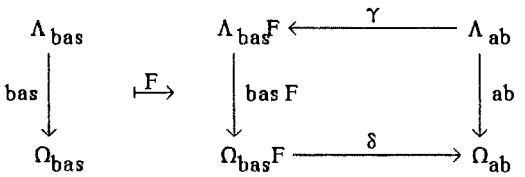
As the example illustrates, it is appropriate to assume that the base consists of a single object bas. In practice, bas will most often be an aggregate object composed of a collection of objects which may interact (i.e. bas is the colimit object of some diagram in OB).

So our problem is the following: given an abstract object ab and a base object bas, what is an implemetation of ab over bas? For notational convenience, we index each item of ab by ab ($\Lambda_{ab}, \Omega_{ab}$ etc.), and similarly for bas and the other objects to follow.

Definition 4.9: Let bas and ab be objects. An *implementation* of ab over bas is given by

(1) a *folding* functor $F: BHV \longrightarrow BHV$, and

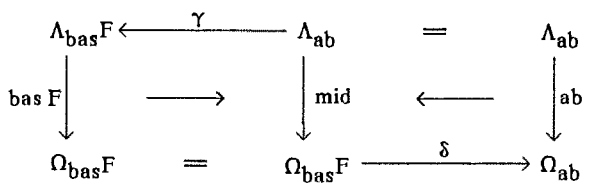
(2) a pair $(\gamma: \Lambda_{ab} \longrightarrow \Lambda_{bas}F,\ \delta: \Omega_{bas}F \longrightarrow \Omega_{ab})$ of behaviour morphisms such that the following diagram in BHV commutes.



That is, the observation $ab(\lambda)$ associated with a life cycle $\lambda \in \Lambda_{ab}$ can be "calculated" using the "encode" and "decode" morphisms γ and δ, respectively, defined on the folded base which has the appropriate transactions in its life cycles.

The difference from the corresponding definition in ES90 is that γ and δ are required to be behaviour morphisms here, not just mappings. As a consequence, ab life cycles are mapped to bas F life cycles "of equal length" (over the same time domain), and correspondingly for δ. That is, an abstract event is mapped to a transaction (which counts as "one step"), and only the observations after completed transactions are shown, not intermediate observations inside a transaction.

The difference between the encode-decode part of an object implementation and an object morphism is that, for the former, the pair of behavior morphisms (γ, δ) is in opposite directions. The question whether implementations can be expressed by morphisms can now be answered easily from the following diagram where $mid = \gamma; bas$.



This diagram shows that the (γ, δ) part of an implementation of ab over bas is the same as two object morphisms, namely $(\gamma, id): bas\,F \longrightarrow mid$ and $(id, \delta): ab \longrightarrow mid$. These two object morphisms deal with event encoding and observation decoding separately (in contrast to the extension/ encapsulation construction in ES90).

5. Concluding Remarks

We have outlined a general categorial framework as a semantic basis for object-oriented approaches. In this paper, we concentrate on single objects and how they are related via interaction, inheritance, etc., and how they are composed to form complex objects. An essential feature of this approach is its generality, especially with respect to the underlying time (or even time-space) domains. This leaves the possibility for incorporating powerful process models, including nondeterminism and forms of concurrency more general than interleaving. So far, however, we have used simple deterministic interleaving models in our examples, albeit with liveness and initiative (SE90). The integration of more powerful models has still to be worked out in detail.

Clearly, the theory has to be extended to cover object types and object classes as well as the various (inheritance) relationships between instances, types and classes.

The development of the semantic domain is being synchronized within the IS-CORE project with work on logic and proof theory for objects (FS90, FSMS90, FM90). It seems that the present general framework is a major step forward towards bringing the semantics and logics of objects together.

Eventually, semantic and logic foundations should prove their usefulness for designing and implementing better languages and systems. Also within the IS-CORE project, work is being carried out towards this end, i.e. developing a broad-spectrum language for object-oriented system specification and development (JSS90, Sa90). A recent overview of object-oriented system development is given in Ve90.

Acknowledgments

Part of this work has been carried out within the ESPRIT-2 BRA 3023 IS-CORE (Information Systems - COrrectness and REusability). The authors are greatly indebted to the other members of IS-CORE, especially Jose-Felix Costa, Cristina Sernadas, Jose Fiadeiro, Miguel Dionisio, Tom Maibaum, Gunter Saake and Ralf Jungclaus for listening to and commenting on the ideas described in this paper.

References

BG77 Burstall,R;Goguen,J.: Putting Theories Together to Make Specifications. Proc. Fifth International Joint Conference on Artificial Intelligence, (R. Reddy, ed.), Dept of Computer Science, Carnegie-Mellon University, 1977, 1045-1058

BG80 Burstall,R;Goguen,J.: The Semantics of Clear, a Specification Language. Proc. 1979 Copenhagen Winter School on Abstract Software Specification, (D. Bjorner, ed.), LNCS 86, Springer Verlag, 1980, 292-332

BT88 Bergstra,J.A.;Tucker,J.V.: The Inescapable Stack: an Exercise in Algebraic Specification with Total Functions. Report No. P8804, Programming Research Group, University of Amsterdam 1988

Di88 Dittrich,K.(ed.): Advances in Object-Oriented Database Systems. LNCS 334, Springer-Verlag, Berlin 1988

ESS88 Ehrich,H.-D.;Sernadas,A.;Sernadas,C.: Abstract Object Types for Databases. In Di88, 144-149

ESS89 Ehrich,H.-D.;Sernadas,A.;Sernadas,C.: Objects, Object Types and Object Identity. Categorical Methods in Computer Science with Aspects from Topology (H. Ehrig et al (eds.), LNCS 393, Springer-Verlag, 142-156

ESS90 Ehrich,H.-D.;Sernadas,A.;Sernadas,C.: From Data Types to Object Types. Journal of Information Processing and Cybernetics EIK 26 (1990) 1/2, 33-48

ES90 Ehrich,H.-D.;Sernadas,A.: Algebraic Implementation of Objects over Objects. Proc. REX Workshop on Stepwise Refinement of Distributed Systems: Models, Formalism, Correctness (J.W. deBakker, W.-P. deRoever, G. Rozenberg, eds.), LNCS 430, Springer-Verlag, Berlin 1990, 239-266

FM90 Fiadeiro,J.;Maibaum,T.: Describing, Structuring and Implementing Objects. This volume.

FS90 Fiadeiro,J.;Sernadas,A.: Logics of Modal Terms for Systems Specification. Journal of Logics and Computation (to appear)

FSMS90 Fiadeiro,J.;Sernadas,C.;Maibaum,T.;Saake,G.: Proof-Theoretic Semantics of Object-Oriented Specification Constructs. Proc. IFIP 2.6 Working Conference DS-4, Windermere (UK) 1990 (to be published by North Holland)

GB84 Goguen,J.A.;Burstall, R.: Some Fundamental Algebraic Tools for the Semantics of Computation. Theor. Comp. Sc. 31 (1984), Part 1: 175-209, Part 2: 263-295

GB90 Goguen,J.A.;Burstall, R.: Institutions, Abstract Model Theory for Specification and Programming. Report ECS-LFCS-90-106, Edinburgh University, 1990 (to appear in JACM)

GG78 Goguen,J.;Ginali,S.: A Categorical Approach to General Systems Theory. Applied General Systems Research (G. Klir, ed.), Plenum, 1978, 257-270

GM82 Goguen,J.A.;Meseguer,J.: Universal Realization, Persistent Interconnection and Implementation of Abstract Modules. Proc. 9th Int. Conf. on Automata, Languages and Programming (M.Nielsen, E.M.Schmidt, eds.), LNCS 140, Springer-Verlag, Berlin 1982, 265-281

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

Go71 Goguen,J.: Mathematical Representation of Hierarchically Organized Systems. Global Systems Dynamics (E. Attinger, ed.), S. Karger, 1971, 112-128

Go73 Goguen,J.: Categorical Foundations for General Systems Theory. Advances in Cybernetics and Systems Research, Transcripta Books, 1973, 121-130

Go75 Goguen,J.: Objects. International Journal of General Systems, 1(4), 1975, 237-243

Go79 Goldblatt,R.: Topoi, the Categorial Analysis of Logic. North-Holland Publ. Comp., Amsterdam 1979

Go89 Goguen,J.: A Categorical Manifesto. Technical Report PRG-72, Programming Research Group, Oxford University, March 1989. To appear in Mathematical Structures in Computer Science.

Go90a Goguen,J.: Sheaf Semantics of Concurrent Interacting Objects, 1990. To appear in Mathematical Structures in Computer Science.

Go90b Goguen,J.: An Algebraic Approach to Refinement. Proc. VDM'90: VDM and Z - Formal Methods in Software Development (D.Bjorner, C.A.R.Hoare, H.Langmaack, eds.), LNCS 428, Springer-Verlag, Berlin 1990, 12-28

Gra65 Gray,J.: Sheaves with values in a category. Topology 3(1965), 1-18

Gro71 Grothendieck,A.: Categories fibrees et descente. Lecture Notes in Mathematics, Volume 224, Springer-Verlag, Berlin 1971, 175-194

GTW78 Goguen,J.A.;Thatcher,J.W.;Wagner,E.G.: An Initial Algebra Approach to the Specification, Correctness, and Implementation of Abstract Data Types. Current Trends in Programming Methodology IV: Data Structuring (R. Yeh, ed.), Prentice Hall, Englewood Cliffs 1978, 80-149

Gu90 Gündel,A.: Compatibility Conditions on Subclasses, Dortmund University (unpublished draft)

HC89 Hayes,F.;Coleman,D.: Objects and Inheritance: An Algebraic View. Technical Memo, HP Labs, Information Management Lab, Bristol 1989

Ho72 Hoare,C.A.R.: Proof of Correctness of Data Representations. Acta Informatica 1 (1972), 271-281

JSS90 Jungclaus,R.;Saake,G.;Sernadas,C.: Using Active Objects for Query Processing. Proc. IFIP 2.6 Working Conference DS-4, Windermere (UK) 1990 (to be published by North Holland)

Sa90 Saake,G.: Descriptive Specification of Database Object Behavior (to appear in Data&Knowledge Engineering, North Holland)

SE90 Sernadas,A.;Ehrich,H.-D.: What is an object, after all ?. Proc. IFIP 2.6 Working Conference DS-4, Windermere (UK) 1990 (to be published by North Holland)

SEC90 Sernadas,A.;Ehrich,H.-D.;Costa,J.-F.: From Processes to Objects. The INESC Journal of Research and Development 1 (1990) 1, 7-27

SFSE89a Sernadas,A.;Fiadeiro,J.;Sernadas,C.;Ehrich,H.-D.: The Basic Building Blocks of Information Systems. Information Systems Concepts: An In-Depth Analysis, E. Falkenberg and P. Lindgreen (eds.), North-Holland 1989, 225-246

SFSE89b Sernadas,A.;Fiadeiro,J.;Sernadas,C.;Ehrich,H.-D.: Abstract Object Types: A Temporal Perspective. Temporal Logic in Specification, B. Banieqbal, H. Barringer and A. Pnueli (eds.), LNCS 398, Springer-Verlag 1989, 324-350

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

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

Ve90 Verharen,E.M.: Object-oriented System Development: An Overview. Technical Report, Infolab, Tilburg University 1990