To appear in *Acta Informatica*, Springer-Verlag

Object Inheritance Beyond Subtyping

J.F.Costa, A.Sernadas & C.Sernadas

INESC & Dept. Matemática - IST
Apartado 10105, 1017 Lisboa Codex, PORTUGAL
{fgc,acs,css}@inesc.pt

Abstract. A categorial semantic domain for objects is presented in order to clarify both aggregation and specialization. Three kinds of specialization are discussed: (1) subtyping (specialization with no side effects and no non-monotonic overriding); (2) monotonic specialization (possibly with side effects but still only with monotonic overriding); and (3) non-monotonic specialization (possibly with side effects and non-monotonic overriding). A sequence of three categories of objects differing only in the morphisms is presented. The first one is used to explain object aggregation (respecting locality through a frame constraint) and the strictest form of specialization (subtyping). The second category is shown to be adequate for explaining specialization with side effects (by relaxing the frame constraint). Finally, the third category supports also non-monotonic overriding, by adopting as morphisms suitable partial morphisms of the second one. All these categories are complete and cocomplete. Colimits are used for explaining the semantics of aggregation and multiple inheritance.

1 Introduction

The advantages of the object-oriented approach to software engineering are by now well understood. Many object-oriented languages, systems and methods have been proposed and some of them are now extensively used. To name just a few, there are programming languages like Smalltalk [Goldberg and Robson 83], C^{++} [Stroustrup 86] and Eiffel [Meyer 88], database systems like GemStone [Butterworth *et al* 91], O_2 [Deux *et al* 91], IRIS [Fishman *et al* 87] and ORION [Kim *et al* 88], development methods like GOOD [Seidewitz and Stark 86], MOOD [Kerth 88] and HOOD [Heitz 88], as well as high-level specification languages [Goguen 91, Jungclaus *et al* 91, SernadasA *et al* 89, SernadasC *et al* 91, SernadasC *et al* 92]. Moreover, some effort has been dedicated to the theoretical foundations of this promising paradigm. The overview [Ehrich *et al* 92] points to several schools of thought in this area: ADT-based like [Goguen 91] and process-based like [Ehrich *et al* 91].

Higher programmer productivity through software reuse is paramount among the claimed advantages of object-orientation. A rich mechanism of inheritance is essential for supporting the envisaged levels of software reuse. By inheritance it is understood the construction facility that allows the introduction of an object class inheriting the definitions of a given class: for instance, SAVINGS_ACCOUNT inherits from ACCOUNT in a banking application. This inheritance may be rather liberal in the sense that it may be partial and some overriding should be possible: for example, some methods of ACCOUNT may be skipped or redefined in SAVINGS_ACCOUNT. From a specification point of view this implies a non-monotonic mechanism: not everything that is said about ACCOUNT carries over to SAVINGS_ACCOUNT.

Most practical languages and systems advocate and support rather liberal forms of inheritance. However, we find the opposite in theoretical work. Indeed, when providing a semantics for object-orientation in general and inheritance in particular, either model-based [Ehrich and SernadasA 91] or theory-based [Fiadeiro *et al* 91], it is much easier to consider only monotonic forms of inheritance that rule out some of the envisaged goodies like overriding. As a typical example we point out most of the work carried out within the ESPRIT BRA Working Group IS-CORE [Ehrich *et al* 90, Ehrich and SernadasA 90, SernadasA and Ehrich 90, SernadasA *et al* 90, Ehrich *et al* 91, Ehrich *et al* 92, Ehrich and SernadasA 91, Costa *et al* 92]. Clearly, non-monotonicity may be added afterwards to a given semantic domain by either using general techniques like those associated to default logics [Lukaszewicz 90, Brass and Lipeck 91] or explaining inheritance at the higher level of presentations.

Herein we show that it is possible to bring full, non-monotonic specialization into a simple model of objects and their communities by adopting the categorial notion of *partial morphism*. Along the way we present a complete semantic domain for objects explaining their interaction and aggregation as well.

For the sake of simplicity, we consider herein only *markovian objects* (that is, objects whose slots are updated by the methods taking into account only the current values of the slots). This restriction is at first glance reasonable on practical grounds: in programming and databases only markovian objects are used. However, non-markovian objects are natural at the early specification stage of software development. Furthermore, non-markovian objects may also appear when hiding some of the slots of an object. Fortunately, it should be straightforward to extend the proposed partial morphism semantics of inheritance to non-markovian objects.

The mechanism of hiding (abstraction, the reverse of specialization) is not discussed at all in this paper because the process model we adopted is not rich enough for that purpose, since it does not support internal non-determinism. *En passant*, we just note that the proposed semantics of inheritance based on partial morphisms should carry over to other richer process models, like *event structures* [Winskel 89].

In section 2, the notion of process and a category of processes are introduced. In section 3, the notion of object and a first category of objects are presented that provide adequate semantics for strict specialization (subtyping) and aggregation. Section 4 introduces a relaxed notion of morphism (dropping the frame rule) that is used to explain the semantics of specialization with side effects. In section 5, the full notion of specialization with overriding is explained using partial morphisms in the previous category of objects. Finally, section 6 discusses multiple specialization.

We make moderate use of the theory of categories. The reader may find all the necessary category-theoretic notions in the book by Jiří Adámek *et al* [Adámek *et al* 90]. See also the appendix for a glossary of the less known categorial concepts used throughout this paper.

2 Object Behaviour

2.1 Processes

We assume that an object is a process (its behaviour) endowed with state-dependent attributes (its slots). The execution of a method corresponds to the happening of an event. Therefore it is essential to adopt some process model.

A rather simple, but adequate for the purpose at hand, model of processes is obtained from the traditional *trace model* (PS model, from *prefix-closed set* [Hoare 85]) by dropping the prefix-closure condition.

Let us consider a chocolate vending-machine MACHINE=μx.coin$\bullet$choc$\bullet$x. The traditional semantics of such a process term is the pair $\langle E,\Lambda\rangle$ with E={coin,choc} and Λ is the prefix-closed subset of E* denoted by the regular expression (coin choc)*(coin + ε). But, this semantics provides no information about the process commitments, namely, that a chocolate must always be delivered by the machine once a coin is inserted.

The idea of quiescence, developed in [Jonsson 85, Jonsson 89] within the context of a logic to reason about specifications, and developed in [Costa 89, Costa and SernadasA 90b] within an algebraic framework, provides a solution for the problem of representing state-dependent commitments in a process. We say that a process is in a *quiescent state* iff it is not willing to produce any action. Returning to our example, the quiescent traces of the MACHINE are those that are defined with an equal number of coin and choc events, ie, the set of traces is given by the regular expression (coin choc)*.

According to the proposed model, which we denote by QS (from *quiescent set*), a process is represented by the set of its quiescent traces. For instance, the pair $\langle${coin,choc}, (coin choc)*$\rangle$ is a QS-process, but not a PS-process, since it is not prefix-closed. Clearly, every PS-process is also a QS-process. Such a process is considered to be without commitments or passive, since it is always quiescent. Those QS-processes that are not PS-processes display some commitments. Indeed, we say that the process $\langle${coin,choc}, (coin choc)*$\rangle$ above is active in the sense that after coin, for instance, it is willing to make choc happen.

Definition 2.1.1. A *process* $P=<E,\Lambda>$ is a pair consisting of a finite set E and a set $\Lambda \subseteq E^*$. ◇

According to the previous definition, only finitary commitments will be considered in this paper (see [Costa and SernadasA 90a] for completeness within $E^* \cup E^\omega$).

The elements of E are called *events* and the elements of Λ *traces*. Given a process $P=<E,\Lambda>$ we refer to E as alph(P) — the *event alphabet* of P — and we refer to Λ as lang(P) — the *traces* of P.

2.2 The Category of Processes

As we shall see below, a morphism between processes is induced by a partial map between event alphabets. For this reason, it is mandatory to recall the category of sets and partial functions.

The category Pfn of *partial functions* is defined as Set but with partial functions as morphisms. It has sets as objects, but more morphisms than Set. Therefore, Set is a wide subcategory of Pfn. The isomorphisms of Pfn are the (total) bijections. Furthermore, $\emptyset$ is a zero object of Pfn (both initial and terminal), and $X \times_{Pfn} Y = X +_{Set} (X \times_{Set} Y) +_{Set} Y$ is, up to an isomorphism, the categorial product of sets X and Y with projections that are partial maps. We can extend the existence of products to the existence of all small limits and so Pfn is small complete. The coproduct of X and Y in Pfn coincides with the coproduct in Set. The existence of coproducts can also be easily extended to the existence of all colimits. Thus Pfn is also finitely cocomplete.

A partial map $h:A \rightarrow B$ can be canonically extended to $h^*:A^* \rightarrow B^*$ as follows: (i) $h^*(\varepsilon)=\varepsilon$ and (ii) $h^*(as)$ = if $h(a)\downarrow$ then $h(a)h^*(s)$ else $h^*(s)$. By $h(a)\downarrow$ we mean that h is defined at a. Otherwise we say that h is undefined and we write $h(a)\uparrow$. The function h^* is in turn extended to sets (and also denoted by h^*) by taking the set of images. Let f be a partial function from a set A to a set B and let g be a partial function from a set B to a set C. Then it is straightforward to prove that $(g \circ f)^* = g^* \circ f^*$. Adapting from [Winskel 84] we propose to the following definition:

Definition 2.2.1. A *process morphism* $h:<E_1,\Lambda_1> \rightarrow <E_2,\Lambda_2>$ is a partial event map $h_E:E_1 \rightarrow E_2$ such that $h_E^*(\Lambda_1) \subseteq \Lambda_2$. ◇

Proposition 2.2.2. The processes and process morphisms constitute a concrete category <Proc,U> over Pfn, where $U:Proc \rightarrow Pfn$ is the forgetful functor that sends each process onto its underlying event alphabet.

Proof: Let $f:P_1 \rightarrow P_2$ and $g:P_2 \rightarrow P_3$ be process morphisms and let $h_E=g_E \circ f_E$. We know that $f_E^*(\Lambda_1) \subseteq \Lambda_2$ and $g_E^*(\Lambda_2) \subseteq \Lambda_3$. Thus $g_E^*(f_E^*(\Lambda_1)) \subseteq \Lambda_3$ and it follows that $(g_E \circ f_E)^*(\Lambda_1) \subseteq \Lambda_3$. Thus, process morphisms compose and, moreover, this composition is associative. It is also trivial to recognize the identity morphisms. The faithful functor is the forgetful functor $U:\mathrm{Proc}\rightarrow\mathrm{Pfn}$ that maps each process onto its event alphabet. ◇

Having established the category of processes we now want to show that it is finitely complete and that limits reflect *parallel composition*: parallel composition without interaction for products and parallel composition with interaction for limits.

The parallel composition of two processes having disjoint alphabets (sharing no events), P||Q, allows P and Q to proceed independently and, additionally, if P can perform an event a and Q can perform an event b then P||Q can perform the joint event $a|b$.

Therefore, the alphabet of P||Q should correspond to the categorial product in Pfn of the alphabets of P and Q, ie,

$$\mathrm{alph}(P\|Q)=\mathrm{alph}(P)\times_{\mathrm{Pfn}}\mathrm{alph}(Q)=\mathrm{alph}(P)+_{\mathrm{Set}}(\mathrm{alph}(P)\times_{\mathrm{Set}}\mathrm{alph}(Q))+_{\mathrm{Set}}\mathrm{alph}(Q)$$

An element $a|b$ of $\mathrm{alph}(P)\times_{\mathrm{Set}}\mathrm{alph}(Q)$ stands for concurrent execution of a by P and b by Q. An element a of alph(P) stands for the execution of a by P while Q remains idle. And an element b of alph(Q) stands for the execution of b by Q while P remains idle. That is, the process P||Q would be able, at each instant, to be involved either in an event of P (while Q remains idle), or in an event of Q (P remaining idle), or in both an event of P and one of Q.

And the traces of P||Q are those traces, over alph(P||Q), that are projected onto traces of the process components, ie,

$$\mathrm{lang}(P\|Q)=\{s\in \mathrm{alph}(P\|Q)^*:\pi_{P_E}{}^*(s)\in \mathrm{lang}(P)\text{ and }\pi_{Q_E}{}^*(s)\in \mathrm{lang}(Q)\}$$

A concrete limit can be constructed in two steps: first, obtain the limit of the underlying diagram in the base category, and, second, provide the initial lift of this underlying limit. The next proposition shows how to build a limit in Proc by lifting from the corresponding source in Pfn, ie, how to calculate the parallel composition of a process community $(\langle E_i,\Lambda_i\rangle)_{i\in I}$.

Proposition 2.2.3. Every Pfn source $(f_{i_E}:E\rightarrow U(\langle E_i,\Lambda_i\rangle))_{i\in I}$, with I a finite set, can be lifted to an initial source in Proc.

Proof: With $\Lambda=\{s\in E^*:\text{ for every } i\in I,\ f_{i_E}{}^*(s)\in\Lambda_i\}$, we trivially recognize that the pair $\langle E,\Lambda\rangle$ is a process and that $(f_i:\langle E,\Lambda\rangle\rightarrow\langle E_i,\Lambda_i\rangle)_{i\in I}$ is a source in Proc. We show that this source is initial. Let $\langle E',\Lambda'\rangle$ be an arbitrary process and $g_E:E'\rightarrow E$ a partial function. We have to prove that if, for every $i\in I$, $f_i\circ g$ is a process morphism then $g:\langle E',\Lambda'\rangle\rightarrow\langle E,\Lambda\rangle$ is a process morphism. But this is true since from $f_{i_E}{}^*(g_E{}^*(\Lambda'))\subseteq\Lambda_i$, for every $i\in I$, follows that $g_E{}^*(\Lambda')\subseteq\Lambda$. ◇

The constructive proof of our last result explains why a process community $(\langle E_i,\Lambda_i\rangle)_{i\in I}$ is represented by the set of traces $\Lambda=\{s\in E^*:\text{ for every } i\in I,\ f_{i_E}{}^*(s)\subseteq\Lambda_i\}$, if we take E to be the resulting alphabet of the whole community.

Proposition 2.2.4. Proc is finitely complete.

Proof: This is a consequence of U being a faithful functor that lifts limits of the complete category Pfn. ◇

Proposition 2.2.5. U lifts limits uniquely.

Proof: U lifts limits. It is straightforward to show that $\langle$Proc,U$\rangle$ is amnestic. Then we conclude that U lifts limits uniquely. ◇

The reason why we need unique lifting is that we want to respect the principle of abstraction of structure: in our concrete category Proc if $Uf:UP\to E$ is a bijection then there can only exist one Λ (=lang(P)) such that $\langle E,\Lambda\rangle$ is a process and f_E:alph(P)$\to$E and f_E^{-1}:E$\to$alph(P) are process morphisms.

Pullbacks are a special kind of limit expressing the (synchronized) composition of two processes, say P and Q, sharing common events. We introduce a middle process that contains only the events that should be shared by the two processes and with the most liberal quiescent traces over the shared alphabet. Let us denote this middle process by R. The following corollary is a characterization of pullbacks up to an isomorphism.

Proposition 2.2.6. The vertex of the pullback (in Proc) of $P\xrightarrow{f}R\xleftarrow{g}Q$ is the process $P\|_R Q$, with $\mathrm{lang}(P\|_R Q)=\{s\in \mathrm{alph}(P\|_R Q)^*:\pi_{PE}^*(s)\in \mathrm{lang}(P) \text{ and } \pi_{QE}^*(s)\in \mathrm{lang}(Q)\}$, where $\mathrm{alph}(P\|_R Q)=\{e\in \mathrm{alph}(P\|Q):f(\pi_{PE}(e))=g(\pi_{QE}(e))\}$. Processes P and Q share the events in $f_E(\mathrm{alph}(P))\cap g_E(\mathrm{alph}(Q))$.

Proof: Given the processes P and Q, $P\|_R Q$ is the domain of the initial lifting of the corresponding pullback source in Pfn. ◇

$$\begin{tikzcd}
& \langle\{a\},\{a\}^*\rangle & \\
\langle\{x,y\},\{\varepsilon,xy\}\rangle \arrow[ur, "x \succ a"] \arrow[rr, phantom, "\textit{pullback}"] & & \langle\{z,w\},\{\varepsilon,zw\}\rangle \arrow[ul, "z \succ a"'] \\
& \langle\{a,y,w,y|w\},\{\varepsilon,ayw,awy,a(y|w)\}\rangle \arrow[ul] \arrow[ur] &
\end{tikzcd}$$

Fig. 2.1. A partial, cooperative parallel composition of two processes.

Consider fig. 2.1. Let us calculate the pullback process $P\|_R Q$, where $R=\langle\{a\},\{a\}^*\rangle$. Note that the up vertex is a process with one event in the alphabet (the event a we want to share) and containing all possible sequences of *a*'s. In this way we ensure that the edges of the peak are indeed process morphisms. The alphabet of the parallel composition synchronizing at a ($=x|z$) will contain all the events of the disjoint parallel composition (the categorial product) except those that are projected onto one of the events of a synchronization pair but not to the other. Hence, in this case, we ruled out, for instance, $x|w$ because it contained one of the events of the synchronization pair $x|z$ but not the other.

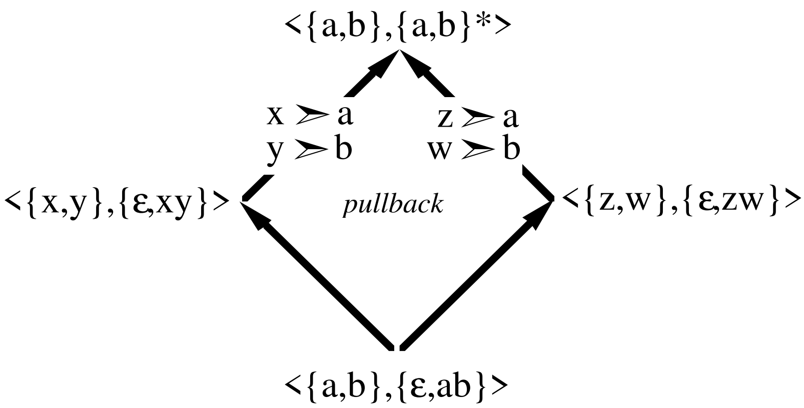

Fig. 2.2. A total, cooperative parallel composition of two processes.

Let us now look at colimits, showing that they reflect choice: disjoint choice for coproducts, ie, disjoint union of traces, and choice with sharing for colimits. In order to apply the technique of lifting colimits of Pfn onto colimits of Proc, we need:

Proposition 2.2.7. Every Pfn sink $(f_{iE}:U(<E_i,\Lambda_i>)\rightarrow E)_{i\in I}$, with I a finite set, can be lifted to a final sink in Proc.

Proof: With $\Lambda=\cup_{i\in I}f_{iE}^*(\Lambda_i)$, we trivially recognise that the pair $<E,\Lambda>$ is a process and that $(f_i:<E_i,\Lambda_i>\rightarrow<E,\Lambda>)_{i\in I}$ is a sink in Proc. We show that this sink is final. Let $<E',\Lambda'>$ be an arbitrary process and $g_E:E\rightarrow E'$ a partial function. We have to prove that if $g \circ f_i$ is a process morphism then g is a process morphism. Since, for every $i\in I$, $g_E^*(f_{iE}^*(\Lambda_i))\subseteq\Lambda'$ we can easily establish that $g_E^*(\Lambda)\subseteq\Lambda'$. ◇

This last constructive proof shows that the choice within a process community $(<E_i,\Lambda_i>)_{i\in I}$ is, as expected, given by the "union" of its traces $\Lambda=\cup_{i\in I}f_{iE}^*(\Lambda_i)$.

Proposition 2.2.8. Proc is finitely cocomplete.

Proof: This is a consequence of U being a faithful functor that lifts all sinks of the cocomplete category Pfn. ◇

Within Proc coproduct stands for disjoint union of alphabets and disjoint union of quiescent traces. Coproduct plus unification leads to general choice. In fig. 2.3 we calculate the binary choice <{x,y},{ε,xy}>$+_{a=x=z}$<{z,w},{ε,zw}> = <{a,y},{ε,ay}>+<{a,w},{ε,aw}> = <{a,y,w},{ε,ay,aw}>

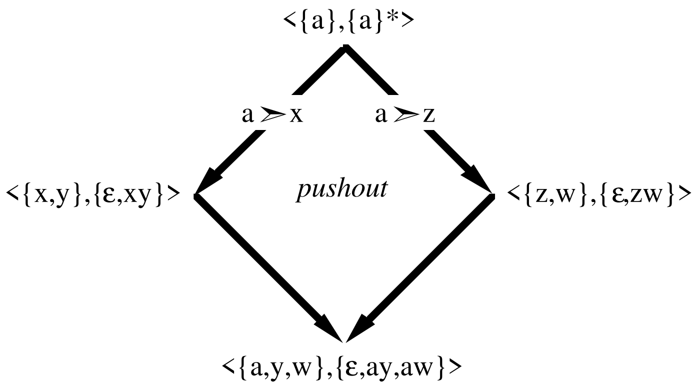

Fig. 2.3. Illustration of choice by pushing out.

3 The Frame Constraint: Subtyping and Aggregation

3.1 Objects

We are now ready to start building the notion of object on top of the notion of process. Let us start with the required slot structure. Let V be a given class of sets.

Definition 3.1.1. A *slot alphabet* is a pair A=<S,cod> where S is a finite set and cod is a total map from S into V. ◇

The idea is that, for each slot a in S, cod returns its codomain, ie, an element of V. In this way cod(a) is the set of the possible values that slot *a* may have. Given a slot alphabet A=<S,cod> we refer to S as slt(A) and we refer to cod as ∂(A). Sometimes, we also write a∈A for a∈slt(A) (mainly, when a∈slt(A) appears in subscript format).

Definition 3.1.2. A *markovian observation structure* over a given event alphabet E is a pair $<A,\{\varphi_{e,a}\}_{e\in E,a\in A}>$ consisting of a slot alphabet A and a E×slt(A)-indexed family φ of *markovian (update) additive functions* $\varphi_{e,a}:\mathbf{2}^{\partial(A)(a)}\rightarrow\mathbf{2}^{\partial(A)(a)}$. ◇

A non-markovian observation structure would imply considering $\{\varphi_{e,a,s}\}_{e\in E,a\in A,s\in E^*}$ instead of $\{\varphi_{e,a}\}_{e\in E,a\in A}$ in order to make the update function also dependent on the state. For instance the object STACK with slot *top* and events *push*, *pop* has a non-markovian observation structure.

Let $<A,\{\varphi_{e,a}\}_{e\in E,a\in A}>$ be a markovian observation structure. Each function $\varphi_{e,a}$ is *additive*, ie, $\varphi_{e,a}(\emptyset)=\emptyset$ and $\varphi_{e,a}(D_1\cup D_2)=\varphi_{e,a}(D_1)\cup\varphi_{e,a}(D_2)$.

If $\varphi_{e,a}(D)$ is empty we say that *the slot a is left undefined after the ocurrence of event e on "state"* D. It may also happen that a slot is given more than one value. Thus, wrt slot values this model allows for *non-determinism*.

Given a markovian observation structure O=<A,$\{\varphi_{e,a}\}_{e\in E,a\in A}$> over E, we refer to A as alph(O) and we refer to $\{\varphi_{e,a}\}_{e\in E,a\in A}$ as map(O).

Clearly, an object should contain a process (its behaviour) and an observation structure (of its slots):

Definition 3.1.3. A *markovian object* <<E,Λ>,<A,$\{\varphi_{e,a}\}_{e\in E,a\in A}$>> is a pair consisting of a process <E,Λ> and a markovian observation structure <A,$\{\varphi_{e,a}\}_{e\in E,a\in A}$> over E. ◇

Given an object ob=<<E,Λ>,<A,$\{\varphi_{e,a}\}_{e\in E,a\in A}$>> let bhv(ob)=<E,Λ> be the behaviour of ob and let ost(ob)=<A,$\{\varphi_{e,a}\}_{e\in E,a\in A}$> be the observation structure of ob.

This definition of markovian object imposes that all slots are independent of each other: the effects of an event upon a slot depend only on the previous value(s) of that slot. However, we can easily work with interdependent attributes by combining them into a single (structured) slot, as illustrated in the following example.

Consider a *bank* ACCOUNT with the following alphabets: $E_{ACCOUNT}$ = {open, close}∪{withdrawal(i), deposit(i): i∈integer}, $S_{ACCOUNT}$ = {red?×balance}. For the sake of simplicity we are considering in the examples that all attributes are integer valued (eg, red?=1 means that the ACCOUNT *is in the red*, that is, with negative balance): therefore, $cod_{ACCOUNT}$(red?×balance)=$\mathbb{Z}\times\mathbb{Z}$. The behaviour of this object is some process over its event alphabet, possibly with $\Lambda_{ACCOUNT}$ denoted by the regular expression open(withdrawal+deposit)*close. Clearly, the object ACCOUNT is markovian with a markovian observation structure given by†

$$\varphi^{ACCOUNT}_{open,red?\times balance} = \lambda x.\{[0,1000]\}$$

$$\varphi^{ACCOUNT}_{deposit(i),red?\times balance} = \lambda x.\{[0,\pi_2(j)+i]:j\in x,\pi_2(j)>-i\}\cup\{[1,\pi_2(j)+i]:j\in x,\pi_2(j)\leq -i\}$$

$$\varphi^{ACCOUNT}_{withdrawal(i),red?\times balance} = \lambda x.\{[0,\pi_2(j)-i]:j\in x,\pi_2(j)>i\}\cup\{[1,\pi_2(j)-i]:j\in x,\pi_2(j)\leq i\}$$

$$\varphi^{ACCOUNT}_{close,red?\times balance} = \lambda x.\{\ \}$$

† In what follows π's stand for projections and inj's for injections.

It is now possible to establish a suitable category of objects by choosing an appropriate notion of object morphism. Such a category is expected to be a concrete category over processes and slot alphabets. Therefore we need:

Definition 3.1.4. A *slot alphabet morphism* $f:A_1 \to A_2$ is a map $f_S:slt(A_1) \to slt(A_2)$ such that $\partial(A_1)(a)=\partial(A_2)(f_S(a))$, for every $a \in slt(A_1)$. ◇

Proposition 3.1.5. The slot alphabets and slot morphisms constitute a complete and cocomplete category Slt (called the *category of slots*).

Proof: Limits and colimits are reflected by the forgetful functor $W:Slt \to Set$ that sends each slot alphabet onto its underlying set of slots. ◇

3.2 Frame Morphisms

With respect to behaviour, we expect that the behaviour of an extended object when restricted to the events of the original object is included in the original behaviour. Therefore, an object morphism should correspond to a process morphism at the behaviour level. On the other hand, with respect to the slots, we want to ensure that if we extend a given object with more slots and events then the additional events should have no effects on the old slots (*no side effects*). Clearly, that should be the case when an object is part of a composite object: the events of the other parts should not affect within the whole the slots of the part. The same is to be expected in the case of a strict form of specialization without side effects: the additional events of the specialization should not affect the inherited slots. Finally, in both cases, the effects of original events on the original slots should be the same in the extended object (*no overriding*). Assuming a *frame constraint* (no side effects and no overriding) we arrive at the following definition:

Definition 3.2.1. A *frame morphism*

$$h:\langle\langle E_1,\Lambda_1\rangle,\langle A_1,\{\varphi^1_{e,a}\}_{e\in E_1,a\in A_1}\rangle\rangle \to \langle\langle E_2,\Lambda_2\rangle,\langle A_2,\{\varphi^2_{e,a}\}_{e\in E_2,a\in A_2}\rangle\rangle$$

is a pair $\langle h_P,h_A\rangle$ where $h_P:\langle E_2,\Lambda_2\rangle \to \langle E_1,\Lambda_1\rangle$ is a process morphism and $h_A:A_1 \to A_2$ is a slot morphism such that the following *observation condition* holds: for every $a \in A_1$, $e \in E_2$, $D \subseteq \partial(A_1)(a)$, if $h_E(e)\downarrow$ then $\varphi^1_{h_E(e),a}(D) \subseteq \varphi^2_{e,h_A(a)}(D)$ else $D \subseteq \varphi^2_{e,h_A(a)}(D)$.◇

From now on, when $h_E(e)\uparrow$ we write $\varphi^1_{h_E(e),a}$ for $id_{\mathbf{2}^{\partial(A_1)(a)}}$. We also write $\varphi^1_{h_E(e),a} \subseteq \varphi^2_{e,h_A(a)}$ for $\varphi^1_{h_E(e),a}(D) \subseteq \varphi^2_{e,h_A(a)}(D)$ for every $a \in A_1$ and $e \in E_2$, $D \subseteq \partial(A_1)(a)$.

Since the morphism condition should reflect the envisaged frame constraint, the reader might expect equality instead of inclusion. However, that would be too strong. The resulting category would not be cocomplete. It is easier to work with the more relaxed notion of morphism and recapture the frame constraint by the interplay between the morphism constraint (inclusion) and the universal property of the colimits as explained below in the case of object aggregation. For specialization, we take advantage of the fact that for functional objects the inclusion degenerates into equality:

Definition 3.2.2. A markovian object $<<E,\Lambda>,<A,\{\varphi_{e,a}\}_{e\in E,a\in A}>>$ is said to be *functional* whenever $\varphi_{e,a}$ maps singletons onto singletons. ◇

Proposition 3.2.3. If h is a frame morphism from a source functional markovian object $<<E_1,\Lambda_1>,<A_1,\{\varphi^1_{e,a}\}_{e\in E_1,a\in A_1}>>$ into a target functional markovian object $<<E_2,\Lambda_2>,<A_2,\{\varphi^2_{e,a}\}_{e\in E_2,a\in A_2}>>$, then $\varphi^1_{h_E(e),a}= \varphi^2_{e,h_A(a)}$, for every $e\in E_2$ and for every $a\in slt(A_1)$. ◇

Proof: Straightforward! Inclusion $\varphi^1_{h_E(e),a}\subseteq\varphi^2_{e,h_A(a)}$ degenerates into equality because $\varphi^1_{h_E(e),a}$ and $\varphi^2_{e,h_A(a)}$ are additive and preserve singletons. ◇

In practice, we tend to define only functional objects. But as we shall see, it is mandatory to work with auxiliary non-functional objects to cope with side effects and overriding. Therefore, we introduce:

Proposition 3.2.4. Markovian objects and frame morphisms constitute a concrete category <FMOb,VF> over $Proc^{op}\times Slt$. ◇

Proof: Object morphisms compose and this composition is associative. Moreover, it is trivial to recognize the identity morphisms. The faithful functor is the forgetful functor VF into $Proc^{op}\times Slt$ such that VF(ob)=<bhv(ob),alph(ost(ob))> on objects and $VF(f:ob_1\rightarrow ob_2)= <f_P,f_A>$ on morphisms. ◇

Now we are interested in the colimits of FMOb since they correspond to limits of the underlying processes. As we shall see in section 4.3, colimits in FMOb correspond to the aggregation of (possibly) interacting objects.

Proposition 3.2.5. The category FMOb has an initial object.

Proof: Take $I=<<\varnothing,\{\varepsilon\}>,<\varnothing,\varnothing>>$. For every object $<<E,\Lambda>,<A,\{\varphi_{e,a}\}_{e\in E,a\in A}>>$ the completely undefined map $\perp_E:E\rightarrow\varnothing$ induces a morphism $\perp:<E,\Lambda>\rightarrow<\varnothing,\{\varepsilon\}>$ because $\perp_E{}^*(\Lambda)\subseteq\{\varepsilon\}$. Taking $\varnothing_S:\varnothing\rightarrow slt(A)$ we see that the observation condition is trivially satisfied. Thus $\perp_E$ and $\varnothing_S$ induce a unique morphism. ◇

Proposition 3.2.6. Every sink

$$(<f_{iP},f_{iS}>:VF(<<E_i,\Lambda_i>,<A_i,\{\varphi^i_{e,a}\}_{e\in E_i,a\in A_i}>>)\rightarrow<<E,\Lambda>,A>)_{i\in I},$$

with I a finite set, can be lifted to a final sink in FMOb.

Proof: Define $ob=<<E,\Lambda>,<A,\{\varphi_{e,a}\}_{e\in E,a\in A}>>$, with $\varphi_{e,a}=\cup_{i\in I}\cup_{x\in f_{iS}{}^{-1}(a)}\varphi^i{}_{f_{iE}(e),x}$. Clearly, $\varphi_{e,a}$ is additive. Moreover, we recognize that

$$(f_i:<<E_i,\Lambda_i>,<A_i,\{\varphi^i_{e,a}\}_{e\in E_i,a\in A_i}>>\rightarrow<<E,\Lambda>,<A,\{\varphi_{e,a}\}_{e\in E,a\in A}>>)_{i\in I}$$

is a sink in FMOb. We show that this sink is final. Let us consider an arbitrary object $<<E',\Lambda'>,<A',\{\varphi'_{e,a}\}_{e\in E',a\in A'}>>$ and a morphism $<g_P,g_S>:<<E,\Lambda>,A>\rightarrow<<E',\Lambda'>,A'>$ in $Proc^{op}\times Slt$. Assuming that $g\circ f_i$ is a morphism we find that, for every $i\in I$,

$\varphi^{i}_{{f_i}_E(gE(e)),a} \subseteq \varphi'_{e,gS({f_i}_S(a))}$. Thus, we have $\varphi_{gE(e),a} = \cup_{i\in I} \cup_{x\in {f_i}_S^{-1}(a)} \varphi^{i}_{{f_i}_E(gE(e)),x} \subseteq \cup_{i\in I} \cup_{x\in {f_i}_S^{-1}(a)} \varphi'_{e,gS({f_i}_S(x))} = \cup_{i\in I} \varphi'_{e,gS(a)} = \varphi'_{e,gS(a)}$. ◇

Proposition 3.2.7. FMOb is finitely cocomplete.

Proof: VF lifts colimits of the cocomplete base category $Proc^{op} \times Slt$. ◇

3.3 Subtyping

At this point we are ready to start the discussion of what is the meaning of ob' *is-a* ob, that is, ob' *is a specialization of* ob. Intuitively, we would like it to be: ob' can be taken as ob if we forget a few details. For instance, a SAVINGS_ACCOUNT is an ACCOUNT in this sense. The former may have a few more slots and events, more specific behaviour and valuation, but it should be possible to forget all that and recognize a simple ACCOUNT.

Therefore, a *strict is-a* relationship (subtyping) should correspond to a frame object morphism in the opposite direction. In this strict sense, ob' *is-a* ob iff there is a *frame inclusion morphism*† from ob into ob'. We classify this relationship as strict because as we shall see it disallows both side effects and overriding.

For instance, we would say that SAVINGS_ACCOUNT *is (strictly) an* ACCOUNT iff there is a frame inclusion from ACCOUNT into SAVINGS_ACCOUNT:

$$
\begin{aligned}
E_{SAVINGS\_ACCOUNT} &= \{open, close, new\_day\} \\
&\quad \cup \{start\_saving, end\_saving\} \\
&\quad \cup \{withdrawal(i), deposit(i) : i \in integer\} \\
S_{SAVINGS\_ACCOUNT} &= \{red? \times balance, saving?, days\}
\end{aligned}
$$

With respect to behaviour, the morphism requirement is very simple and natural: any possible trace of SAVINGS_ACCOUNT should degenerate into a possible trace of ACCOUNT when we forget the specific events. Note that we are free to impose further constraints on the behaviour of SAVINGS_ACCOUNT even with respect to the events inherited from ACCOUNT. For instance, we might impose that if the SAVINGS_ACCOUNT is in the saving mode (ie, if saving?=1) then no withdrawals are allowed (only deposits). But, in order to illustrate another aspect discussed below, we prefer instead to allow a 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.

† By frame inclusion morphism f from ob into ob' we mean a morphism induced by a surjective partial inclusion on events and an inclusion on slots.

The slot 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 slot saving? indicates if the ACCOUNT is in the saving mode (saving?=1) or not (saving?=0).

$$\varphi^{SAVINGS\_ACCOUNT}{}_{open,saving?} = \lambda x.\{0\}$$
$$\varphi^{SAVINGS\_ACCOUNT}{}_{deposit(i),saving?} = \lambda x.x$$
$$\varphi^{SAVINGS\_ACCOUNT}{}_{withdrawal(i),saving?} = \lambda x.\{0\}$$
$$\varphi^{SAVINGS\_ACCOUNT}{}_{start\_saving,saving?} = \lambda x.\{1\}$$
$$\varphi^{SAVINGS\_ACCOUNT}{}_{end\_saving,saving?} = \lambda x.\{0\}$$
$$\varphi^{SAVINGS\_ACCOUNT}{}_{new\_day,saving?} = \lambda x.x$$

$$\varphi^{SAVINGS\_ACCOUNT}{}_{open,days} = \lambda x.x$$
$$\varphi^{SAVINGS\_ACCOUNT}{}_{deposit(i),days} = \lambda x.x$$
$$\varphi^{SAVINGS\_ACCOUNT}{}_{withdrawal(i),days} = \lambda x.x$$
$$\varphi^{SAVINGS\_ACCOUNT}{}_{start\_saving,days} = \lambda x.\{0\}$$
$$\varphi^{SAVINGS\_ACCOUNT}{}_{end\_saving,days} = \lambda x.x$$
$$\varphi^{SAVINGS\_ACCOUNT}{}_{new\_day,days} = \lambda x.\{j+1 : j \in x\}$$

Note that the original events of ACCOUNT are allowed to change the specific slots of SAVINGS_ACCOUNT — namely withdrawals may change the extra slots.

On the contrary, the new events of SAVINGS_ACCOUNT do not affect the values of the old slots inherited from ACCOUNT (no side effects!) because of the frame morphism constraint on functional objects. For instance:

$$\varphi^{SAVINGS\_ACCOUNT}{}_{start\_saving,red?\times balance} = \lambda x.x$$
$$\varphi^{SAVINGS\_ACCOUNT}{}_{end\_saving,red?\times balance} = \lambda x.x$$

Moreover, the effects of the old events on the old slots inherited from ACCOUNT are the same in SAVINGS_ACCOUNT (no overriding!) again because of the frame morphism constraint on functional objects. For instance:

$$\varphi^{SAVINGS\_ACCOUNT}{}_{withdrawal(i),red?\times balance} = \lambda x.\{[0,\pi_2(j)-i] : j \in x, \pi_2(j) > i\} \cup \{[1,\pi_2(j)-i] : j \in x, \pi_2(j) \leq i\}$$

$$\varphi^{SAVINGS\_ACCOUNT}{}_{deposit(i),red?\times balance} = \lambda x.\{[0,\pi_2(j)+i]:j\in x\wedge\pi_2(j)>-i\}\cup \{[1,\pi_2(j)+i]:j\in x\wedge\pi_2(j)\leq -i\}$$

Such a strict version of the *is-a* relationship (subtyping), although useful in some cases, has been widely criticized as being insufficient, but we shall provide more liberal versions in the subsequent sections of this paper.

3.4 Aggregation

We just saw that frame morphisms explain only a rather strict form of specialization: subtyping. Therefore, for inheritance purposes we need to consider other more general morphisms. However, the frame morphisms are precisely what is needed to explain object aggregation! Indeed, the slots of an aggregation component should not be affected by the events of the other parts (no side effects!). And the effects of the events of an aggregation component should be the same in the whole (no overriding!).

Therefore, we are now ready to characterize object *aggregation* (without and with interaction) using some colimits in the category FMOb, keeping in mind that wrt behaviour it should correspond to parallel composition. For instance, binary coproducts provide the disjoint aggregation of the argument objects: disjoint parallel composition of behaviours, plus addition of valuation maps.

As an illustration consider the coproduct of two bank accounts (two isomorphic copies of ACCOUNT): $ACCOUNT_1\|ACCOUNT_2$. Note that we are extending the traditional notation for parallel composition of processes to object aggregation, since they are so closely related. As expected these two independent objects can be put together without affecting each other. Clearly, there must be a frame injection morphism from $ACCOUNT_i$ into $ACCOUNT_1\|ACCOUNT_2$, for i=1,2.

Now consider a CLIENT owning an ACCOUNT: E_{CLIENT} ={become_client_request, req_refusal, open_acc, close_acc}∪{become_client(i), withdrawal(i), deposit(i): i∈integer}, S_{CLIENT} ={number}, and L_{CLIENT} denoted by the regular expression

become_client_request req_refusal +
become_client_request become_client(99) open_acc(withdrawal + deposit)*close_acc

The markovian observation structure of CLIENT is given by:

$$\varphi^{CLIENT}{}_{become\_client\_request,number} = \lambda x.x$$

$\varphi^{CLIENT}{}_{req\_refusal,number}$	=	$\lambda x.x$
$\varphi^{CLIENT}{}_{open\_account,number}$	=	$\lambda x.x$
$\varphi^{CLIENT}{}_{close\_account,number}$	=	$\lambda x.x$
$\varphi^{CLIENT}{}_{become\_client(i),number}$	=	$\lambda x.\{i\}$
$\varphi^{CLIENT}{}_{withdrawal(i),number}$	=	$\lambda x.x$
$\varphi^{CLIENT}{}_{deposit(i),number}$	=	$\lambda x.x$

We want to establish their joint behaviour in the presence of some interaction. For instance, the ACCOUNT is opened by the CLIENT. Assuming that the objects ACCOUNT and CLIENT are as specified before, we would like to impose the following set of interaction equations:

$$\rho = \{ \begin{array}{rcl} \text{CLIENT.open\_acc} & = & \text{ACCOUNT.open,} \\ \text{CLIENT.deposit(i)} & = & \text{ACCOUNT.deposit(i),} \\ \text{CLIENT.withdrawal(i)} & = & \text{ACCOUNT.withdrawal(i),} \\ \text{CLIENT.close\_acc} & = & \text{ACCOUNT.close} \end{array} \}$$

That is, open, deposit, close and withdrawal actions are to be shared by the two objects. The aggregation $\text{CLIENT}\|_\rho\text{ACCOUNT}$ 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* $\text{CLIENT}\|_\rho\text{ACCOUNT}$.

Another form of interaction can be achieved by *slot sharing*. However it should be stressed that, in order to obtain the envisaged joint behaviour, all the events affecting the slot must also be shared. That is, it is only possible to share whole objects. This seems to be essential to object orientation.

The proposition 3.6 and the constructive proof of proposition 4.1.6 show how we can aggregate objects even in the presence of memory sharing.

Proposition 3.4.1. The FMOb-pushout vertex of $ob_1 \xleftarrow{f} ob \xrightarrow{g} ob_2$ is $ob_1\|_{ob}ob_2 = \langle bhv(ob_1)\|_{bhv(ob)}bhv(ob_2), \langle A_1 \vee A_2, \{\varphi_{e,a}\}_{e\in E_1\wedge E_2, a\in A_1\vee A_2}\rangle\rangle$, where $A_1\vee A_2$ is the Slt-pushout vertex of $A_1 \xleftarrow{fS} A \xrightarrow{gS} A_2$, $E_1\wedge E_2$ is Pfn-pullback vertex of $E_1 \xrightarrow{fE} E \xleftarrow{gE} E_2$ and, for $e\in E_1\wedge E_2$, $a\in A_1\vee A_2$, $\varphi_{e,a} = \cup_{x\in inj1_S{}^{-1}(a)}\varphi^1{}_{\pi 1_E(e),x} \cup \cup_{x\in inj2_S{}^{-1}(a)}\varphi^2{}_{\pi 2_E(e),x}$. ◇

Clearly, pushouts in FMOb reflect pullbacks in Proc (pushouts in $Proc^{op}$) and pushouts in Slt. Moreover, if I is the initial FMOb object then $ob_1\|_I ob_2$ stands for the aggregation (parallel composition) of the two independent objects ob_1 and ob_2.

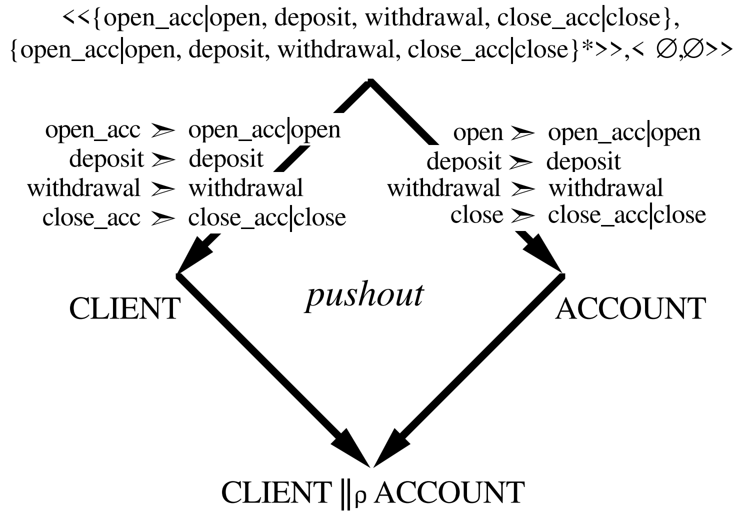

Fig. 3.1. Aggregation of CLIENT and ACCOUNT sharing open_acc|open, deposit(i) for every i, withdrawal(i) for every i, and close_acc|close.

In CLIENT$\|_\rho$ACCOUNT events like deposit or withdrawal leave the slot number of CLIENT unchanged. We have, for example,

$$\varphi^{\mathrm{CLIENT}\|_\rho\mathrm{ACCOUNT}}{}_{\mathrm{deposit(i),number}}$$

$$=\varphi^{\mathrm{CLIENT}}{}_{\mathrm{deposit(i),number}}\cup\bigcup_{x\in \mathrm{inj}_2{}^{-1}(\mathrm{number})}\varphi^{\mathrm{ACCOUNT}}{}_{\mathrm{deposit(i)},x}$$

$$=\varphi^{\mathrm{CLIENT}}{}_{\mathrm{deposit(i),number}}\cup\bigcup_{x\in\emptyset}\varphi^{\mathrm{ACCOUNT}}{}_{\mathrm{deposit(i)},x}$$

$$=\varphi^{\mathrm{CLIENT}}{}_{\mathrm{deposit(i),number}}$$

Interaction between objects means sharing either events or slots and their related events, ie, either event sharing or memory sharing. Both can be accomplished using diagrams of frame injection morphisms† lifted from $\mathrm{Proc}^{\mathrm{op}}\times\mathrm{Slt}$. Clearly, event and slot sharing are special cases of object sharing. For event sharing we take an empty set of slots in the middle object. Fig. 4.3.1 depicts the pushout corresponding to the aggregation of CLIENT and ACCOUNT sharing the object induced by the interaction set ρ above.

It should be stressed that the aggregation (without or with interaction) of two functional objects is still functional. Anyway, the universal property of the aggregation also implies

† By frame injection morphism f from ob into ob' we mean a morphism induced by a surjective partial injection on events and an injection on slots.

that the frame morphism constraint degenerates into equality. That is, we have as intended no side effects and no overriding in aggregations.

4 Specialization with Side Effects

4.1 Dropping the Frame Constraint

In order to allow side effects we propose the following definition of object morphism (relaxing the frame constraint):

Definition 4.1.1. An *object morphism*

$$h:<<E_1,\Lambda_1>,<A_1,\{\varphi^1{}_{e,a}\}_{e\in E_1,a\in A_1}>>\to<<E_2,\Lambda_2>,<A_2,\{\varphi^2{}_{e,a}\}_{e\in E_2,a\in A_2}>>$$

is a pair $<h_P,h_A>$ where $h_P:<E_2,\Lambda_2>\to<E_1,\Lambda_1>$ is a process morphism and $h_A:A_1\to A_2$ is a slot morphism such that the following condition holds: for every $a\in A_1$, $e\in E_2$ such that $h_E(e)\downarrow$, $\varphi^1{}_{h_E(e),a}\subseteq\varphi^2{}_{e,h_A(a)}$. ◇

Proposition 4.1.2. Markovian objects and object morphisms constitute a concrete category <MOb,V> over $Proc^{op}\times Slt$. Moreover, FMOb is a wide subcategory of MOb.

Proof: Object morphisms compose and this composition is associative. Moreover, it is trivial to recognize the identity morphisms. The faithful functor is the forgetful functor V into $Proc^{op}\times Slt$ such that V(ob)=<bhv(ob),alph(ost(ob))> on objects and $V(f:ob_1\to ob_2)=<f_P,f_A>$ on morphisms. ◇

Proposition 4.1.3. Every sink

$(<f_{iP},f_{iS}>:V(<<E_i,\Lambda_i>,<A_i,\{\varphi^i{}_{e,a}\}_{e\in E_i,a\in A_i}>>)\to<<E,\Lambda>,A>)_{i\in I}$, with I a finite set, can be lifted to a final sink in MOb.

Proof: Define $ob=<<E,\Lambda>,<A,\{\varphi_{e,a}\}_{e\in E,a\in A}>>$, with $\varphi_{e,a}=\cup_{i\in J}\cup_{x\in f_{iS}^{-1}(a)}\varphi^i{}_{f_{iE}(e),x}$ where $J=\{i\in I:f_{iE}(e)\downarrow\}$. The proof is similar to the proof of proposition 4.1.6. ◇

Proposition 4.1.4. MOb is finitely cocomplete.

Proof: V is a forgetful functor that lifts colimits of the cocomplete base category $Proc^{op}\times Slt$. ◇

Let $(<f_{iP},f_{iS}>:V(<<E_i,\Lambda_i>,<A_i,\{\varphi^i{}_{e,a}\}_{e\in E_i,a\in A_i}>>)\to<<E,\Lambda>,A>)_{i\in I}$, with I a finite set, be a sink in $Proc^{op}\times Slt$. Define the object $ob=<<E,\Lambda>,<A,\{\varphi_{e,a}\}_{e\in E,a\in A}>>$ with $\varphi_{e,a}=\cup_{i\in I}\cup_{x\in f_{iS}^{-1}(a)}\varphi^i{}_{f_{iE}(e),x}$. Define the object $ob'=<<E,\Lambda>,<A,\{\varphi'_{e,a}\}_{e\in E,a\in A}>>$ with $\varphi'_{e,a}=\cup_{i\in J}\cup_{x\in f_{iS}^{-1}(a)}\varphi^i{}_{f_{iE}(e),x}$ where J is the set $\{i\in I:f_{iE}(e)\downarrow\}$. We know already that $(f_i:<<E_i,\Lambda_i>,<A_i,\{\varphi^i{}_{e,a}\}_{e\in E_i,a\in A_i}>>\to ob')_{i\in I}$ is a final sink in MOb. We have a morphism $\rho:ob'\to ob$ in MOb induced by $<id_E,id_S>$. Thus the final sinks in FMOb and MOb are related through ρ, ie, $(f_i:<<E_i,\Lambda_i>,<A_i,\{\varphi^i{}_{e,a}\}_{e\in E_i,a\in A_i}>>\to ob)_{i\in I}$ is a final sink in FMOb whenever $(f_i:<<E_i,\Lambda_i>,<A_i,\{\varphi^i_{e,a}\}_{e\in E_i,a\in A_i}>>\to ob')_{i\in I}$ is a final sink in MOb.

Fortunately, MOb is also finitely complete. Limits within MOb are also important for overriding as we shall see in section 6.1.

Proposition 4.1.5. Every source

$(<f_{iP},f_{iS}>:<<E,\Lambda>,A>\to V(<<E_i,\Lambda_i>,<A_i,\{\varphi^i_{e,a}\}_{e\in E_i,a\in A_i}>>))_{i\in I}$, with I a finite set, can be lifted to an initial source in MOb.

Proof: Define $ob=<<E,\Lambda>,<A,\{\varphi_{e,a}\}_{e\in E,a\in A}>>$, with $\varphi_{e,a}=\cap_{i\in I}\cap_{x\in f_{iE}^{-1}(e)}\varphi^i_{x,f_{iS}(a)}$. Clearly, $\varphi_{e,a}$ is additive. Moreover, we recognize that

$$(f_i:<<E,\Lambda>,<A,\{\varphi_{e,a}\}_{e\in E,a\in A}>>\to<<E_i,\Lambda_i>,<A_i,\{\varphi^i_{e,a}\}_{e\in E_i,a\in A_i}>>)_{i\in I}$$

is a source in MOb. We show that this source is initial. Consider an arbitrary object $<<E',\Lambda'>,<A',\{\varphi'_{e,a}\}_{e\in E',a\in A'}>>$ and a morphism $<g_P,g_S>:<<E',\Lambda'>,A'>\to<<E,\Lambda>,A>$ in $Proc^{op}\times Slt$. Assuming that $f_i\circ g$ is a morphism we find that, for every $i\in I$, $\varphi'_{gE(f_{iE}(e)),a}\subseteq\varphi^i_{e,f_{iS}(gS(a))}$. Thus, we have $\varphi_{e,gS(a)}=\cap_{i\in I}\cap_{x\in f_{iS}^{-1}(e)}\varphi^i_{x,f_{iS}(gS(a))}\supseteq\cap_{i\in I}\cap_{x\in f_{iE}^{-1}(e)}\varphi'_{gE(f_{iE}(x)),a}=\cap_{i\in I}\varphi'_{gE(e),a}=\varphi'_{gE(e),a}$. ◇

Proposition 4.1.6. MOb is finitely complete.

Proof: V lifts limits of the complete base category $Proc^{op}\times Slt$. ◇

4.2 Specialization and Side Effects

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 (side effect of start_saving on balance). We would like to impose

$$\varphi^{SAVINGS\_ACCOUNT}_{start\_saving,red?\times balance}=\lambda x.\{[0,\pi_2(j)\text{-}50]:j\in x,\pi_2(j)>50\}\cup\{[1,\pi_2(j)\text{-}50]:j\in x,\pi_2(j)\leq 50\}$$

The existence of an *inclusion* in MOb from ACCOUNT into SAVINGS_ACCOUNT tells us that if we choose a slot *a* in $S_{ACCOUNT}$ and an inherited event *e* in $E_{SAVINGS\_ACCOUNT}$ then $\varphi^{ACCOUNT}_{e,a}=\varphi^{SAVINGS\_ACCOUNT}_{e,a}$. However, nothing is said about what happens if we choose a new event of $E_{SAVINGS\_ACCOUNT}$. Thus morphisms in MOb are particularly suitable for providing the semantics of inheritance (side effects being possible).

With respect to behaviour, the morphism condition remains as discussed in FMOb: any possible trace of SAVINGS_ACCOUNT should degenerate into a possible trace of ACCOUNT when we forget the new events of SAVINGS_ACCOUNT.

5 Method Overriding

As we just saw, the morphisms of MOb are liberal enough to explain specialization with side effects. But they completely disallow non-monotonic overriding in the sense that the effects of inherited events on inherited slots are not changed at all. In order to cope with non-monotonic overriding we need to consider even more relaxed morphisms. Since the effects in the specialization may be rather different from those in the original object, a decoupling is needed between the two objects, suggesting the use of partial morphisms. Indeed, a partial morphism from ob into ob' is a morphism from a subobject of ob into ob'. That subobject characterizes what is kept from ob in ob'. The rest may be changed (overriden).

5.1 Partial Morphisms

Definition 5.1.1. Let B be the class of $Pfn^{op}\times Slt$ identity morphisms. A 2-source in MOb, $ob_1 \xleftarrow{\iota} x \xrightarrow{f} ob_2$, with $\iota \in B$ is called a *partial markovian morphism* from ob_1 into ob_2. ◇

For a partial morphism $ob_1 \xleftarrow{\iota} x \xrightarrow{f} ob_2$ within MOb we just write $ob_1 \leftarrow x \xrightarrow{f} ob_2$ or even $f: ob_1 \nrightarrow ob_2$.

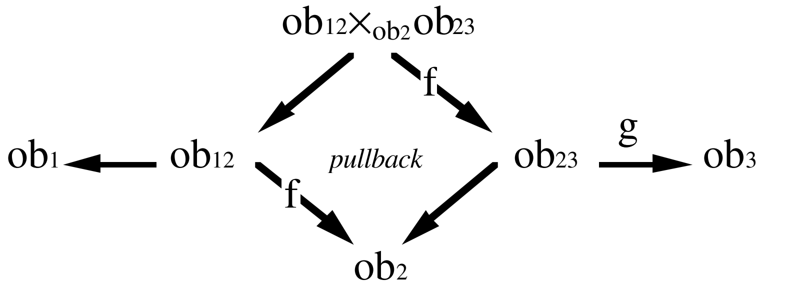

Fig. 5.1. Composition of partial morphisms.

Let us now define *composition of partial morphisms*. But first recall that $x_1 \leftarrow x_1 \xrightarrow{f} x_2$ is a pullback of $x_1 \xrightarrow{f} x_2 \leftarrow x_2$ and $x_1 \rightarrow x_1 \xleftarrow{f} x_2$ is a pushout of $x_1 \xleftarrow{f} x_2 \rightarrow x_2$. Thus, if $ob_1 \leftarrow ob_{12} \xrightarrow{f} ob_2$ and $ob_2 \leftarrow ob_{23} \xrightarrow{g} ob_3$ are partial morphisms within MOb, then a pullback of $ob_{12} \xrightarrow{f} ob_2 \leftarrow ob_{23}$ can be obtained by lifting from the source $<E_{12},A_{12}> \leftarrow <E_{12},A_{12}> \xrightarrow{f} <E_{23},A_{23}>$ in $Pfn^{op}\times Slt$. Let $ob_{12}\times_{ob_2}ob_{23}$ be the vertex of the corresponding initial source in MOb.

Definition 5.1.2. Composition of two partial morphisms $ob_1 \leftarrow ob_{12} \xrightarrow{f} ob_2$ and $ob_2 \leftarrow ob_{23} \xrightarrow{g} ob_3$ is the partial morphism $ob_1 \leftarrow ob_{12} \times_{ob_2} ob_{23} \xrightarrow{g \circ f} ob_3$. ◇

Proposition 5.1.3. Markovian objects and partial markovian morphisms constitute a cocomplete category PMOb. ◇

This result becomes relevant only when expaining multiple inheritance (cf section 6), since aggregation is always carried out within FMOb.

$$\mathrm{FMOb} \hookrightarrow \mathrm{MOb} \hookrightarrow \mathrm{PMOb}$$

Fig. 5.2. The net of object categories.

5.2 The Semantics of Overriding

We already saw an example of a rather restricted form of overriding: the withdrawal event of the ACCOUNT object is redefined to some extent in the SAVINGS_ACCOUNT object. Indeed, in the latter a withdrawal has more effects than in the former — namely, it also affects the values of the additional slot saving?.

The semantics of specialization given above within MOb does allow for this kind of event overriding: an event when inherited may have further effects, but the effects on the inherited slots must remain the same. We could say that we have so far only *monotonic* overriding.

Any stronger form of specialization will also require a more liberal overriding mechanism, allowing, for instance, a different treatment of the inherited slots by the inherited events in the specialized object. In order to provide the semantics of such a *non-monotonic overriding* mechanism we have to consider morphisms in PMOb.

A partial morphism h:ACCOUNT$\not\to$SAVINGS_ACCOUNT introduces a *subobject* ACCOUNT' of ACCOUNT that coincides with ACCOUNT with the exception of the maps $\varphi_{e,a}$ corresponding to the method we may want to override. In ACCOUNT' we must impose $\varphi_{e,a}=\lambda x.\varnothing$ for those maps: then, the maps $\varphi^{ACCOUNT}{}_{e,a}$ and $\varphi^{SAVINGS\_ACCOUNT}{}_{e,a}$ are no more the same.

Therefore, if SAVINGS_ACCOUNT *is-an* ACCOUNT, that is, if there is a partial morphism h:ACCOUNT$\not\to$SAVINGS_ACCOUNT, then we may have method overriding: whenever some rules of ACCOUNT are to be rewritten in SAVINGS_ACCOUNT, the partial morphism ACCOUNT$\leftarrow$ACCOUNT'$\xrightarrow{h}$SAVINGS_ACCOUNT should be chosen so that $\varphi_{e,a}=\lambda x.\varnothing$ in ACCOUNT' for each method *e* to be overridden.

It should be stressed that both source and target of a partial morphism can be functional. However, the auxilliary middle object fails to be functional! In practice, we can go on defining only functional objects (and imposing configurations based upon inclusions and injections of MOb) if we disregard such auxilliary objects.

6 Multiple inheritance

The semantic domain we have been discussing for object aggregation and specialization (possibly, with side effects and non-monotonic overriding) is powerful enough to clarify also the semantics of multiple inheritance.

For illustration, consider a specialization FOREIGN_CURRENCY_ACCOUNT of ACCOUNT. For the sake of generality, assume that it is obtained with side effects and non-monotonic overriding.

$$\begin{tikzcd}
& \text{ACCOUNT} \arrow[dl] \arrow[dr] & \\
\text{SAVINGS_ACCOUNT} \arrow[dr] \arrow[rr, phantom, "\textit{pushout}"'] & & \text{FOREIGN_CURRENCY_ACCOUNT} \arrow[dl] \\
& \text{SAVINGS_FOREIGN_CURRENCY_ACCOUNT} &
\end{tikzcd}$$

Fig. 6.1. Illustration of multiple inheritance as a pushout in PMOb.

We may then be interested in a double specialization of both SAVINGS_ACCOUNT and FOREIGN_CURRENCY_ACCOUNT. To this end, it is useful to consider the pushout in PMOb depicted in fig. 6.1. The object SAVINGS_FOREIGN_CURRENCY_ACCOUNT is the *minimal specialization* of SAVINGS_ACCOUNT and FOREIGN_CURRENCY_ACCOUNT. Therefore, it inherits events, behaviour, slots and effects from both of them, respecting the common ancestor ACCOUNT. But, being minimal, it does not introduce any other events or slots, it does not further restrict the behaviour and it does not introduce more effects. Namely, for instance, the effect of a specific event of SAVINGS_ACCOUNT on a specific slot of FOREIGN_CURRENCY_ACCOUNT is undefined.

Afterwards it is possible to specialize SAVINGS_FOREIGN_CURRENCY_ACCOUNT in order to obtain any desired multiple specialization of both SAVINGS_ACCOUNT and SAVINGS_FOREIGN_CURRENCY_ACCOUNT.

It should be evident that the pushout is in general non-functional even if the given nodes are functional. But, in practice, we should look at the pushout as an auxilliary object upon which a further specialization (possibly with overriding) allows us to return to the functional world.

7 Concluding Remarks

We presented a categorial semantic domain for objects that clarifies both aggregation (with and without interaction) and specialization. We discussed three kinds of specialization: (1) subtyping (specialization with no side effects and no non-monotonic overriding); (2) monotonic specialization (possibly with side effects but still only with monotonic overriding); and (3) non-monotonic specialization (possibily with side effects and non-monotonic overriding). We established three main categories of objects: FMOb, MOb and PMOb. They differ only in the morphisms. FMOb explains object aggregation (respecting locality through a frame constraint) and the strictest form of specialization (subtyping). MOb is adequate for explaining specialization with side effects (by relaxing the frame constraint in the morphism definition). Finally, PMOb supports also non-monotonic overriding by adopting as morphisms suitable partial morphisms of MOb. All these categories are complete and cocomplete. Colimits are used both in the semantics of aggregation and multiple inheritance. The completeness of MOb is essential for setting-up PMOb.

It should be clear from the presentation that the chosen process model (the category Proc) works like a parameter: we may adopt a quite different process model and replicate the construction over it of the relevant categories of objects. That is, the proposed semantics of specialization and aggregation should carry over without problems to any other process model. We adopted a modified trace model because it is enough for the purpose at hand and it is very simple to present. Any other model would require a lot of technical effort just to present it, leaving little space for the main goal of the paper.

Also for the sake of simplicity we concentrated on markovian objects (objects with the property that slot updates depend only on the current values of the slots). We do not expect great difficulties in extending the results to non-markovian objects.

It should be stressed that in this paper we do not distinguish between objects and aspects, contrarily to what is proposed in [Ehrich and SernadasA 91, Ehrich *et al* 92]. Therefore, we ignore the problem of having the same object with different aspects (ie, the same savings account x with several aspects, such as x.ACCOUNT and x.SAVINGS_ACCOUNT). According to the terminology in [Ehrich and SernadasA 91, Ehrich *et al* 92], we may look at this paper as a study of object template inheritance and aggregation. An immediate research concern is to bring the results presented herein to bear on a suitable framework of object identifiers, aspects and classes. In that direction we should stress that we recognise two main steps in the specification of a community of objects: (1) the description of the inheritance network (using, for instance, the morphisms of PMOb); and (2) the description of the configuration/interaction net of the community (using morphisms of FMOb).

We hope that the proposed approach to the semantics of non-monotonic specialization (through partial morphisms of the category used for monotonic specialization) may be of use in other contexts, namely when setting-up an institution for a non-monotonic specification logic from a given institution for a (monotonic) specification logic. This seems to be an interesting area for future research.

Acknowledgements

This work was partially supported by the JNICT Project PMCT/C/TIT/178/90 (FAC3), by the Esprit Basic Research Action 6071 (IS-CORE), and, last but not least, by the ESDI Project OBLOG. The authors are grateful to their colleagues in these three projects for many rewarding interactions. They are also indebted to the anonymous referees for their suggestions towards improving the presentation of the paper.

References

[Adámek *et al* 90]
J.Adámek, H.Herrlich and G.Strecker, *Abstract and Concrete Categories*, Wiley, 1990

[Butterworth *et al* 91]
P.Butterworth, A.Otis and J.Stein, "The GemStone Object Database Management System", *Comm. ACM 34*, 1991, 64-77

[Brass and Lipeck]
S.Brass and U.Lipeck, "Semantics of Inheritance in Logical Object Specifications", *Proc. of the 2nd International Conference on Deductive and Object-Oriented Databases* (DOOD'91), 1991, *to appear*

[Costa 89]
J.F.Costa, *Teoria Algébrica dos Processos Animados*, MSc Thesis, available as a technical report, Universidade Técnica de Lisboa, September 1989

[Costa and SernadasA 90a]
J.F.Costa and A.Sernadas, "Process Models within a Categorial Framework", Research Report, INESC, 1990 (submitted)

[Costa and SernadasA 90b]
J.F.Costa and A.Sernadas, "Progress Assumption in Concurrent Systems", Research Report, INESC, 1990 (submitted)

[Costa *et al* 92]
J.F.Costa, A.Sernadas, C.Sernadas and H.-D.Ehrich, "Object Interaction", Mathematical Foundations of Computer Science, LNCS, Springer-Verlag, 1992, *in print*

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

[Deux *et al* 91]
O.Deux et al, "The O_2 System", *Comm. ACM 34*, 1991, 35-48

[Ehrich *et al* 90]
H.-D.Ehrich, A.Sernadas and C.Sernadas "From Data Types to Object Types", in *Journal of Information Processing and Cybernetics*, EIK 26(1/2), 1990, 33-48

[Ehrich *et al* 91]
H.-D.Ehrich, J.Goguen and A.Sernadas, "A Categorial Theory of Objects as Observed Processes", in J.W. deBakker, W.P.deRoever and G.Rozenberg (eds), *Proc. of the REX90/Workshop on Foundations of Object-Oriented Languages*, LNCS 489, Springer-Verlag, 1991, 203-228

[Ehrich *et al* 92]
H.-D.Ehrich, M.Gogolla and A.Sernadas, "Objects and Their Specifications", in M.Bidoit and C.Choppy (eds), *Recent Trends in Data Type Specification: 8th Workshop on Specification of Abstract Data Types — Selected Papers*, LNCS, Springer-Verlag, 1992, *in print*

[Ehrich and SernadasA 90]
H.-D.Ehrich and A.Sernadas, "Algebraic Implementation of Objects over Objects", in J. deBakker, W.-P.deRoever and G.Rozenberg (eds), *Proc. of the REX Workshop on Stepwise Refinement of Distributed Systems: Models, Formalisms, Correctness*, Springer-Verlag, 1990, 239-266

[Ehrich and SernadasA 91]
H.-D.Ehrich and A.Sernadas, "Object Concepts and Constructions", in G.Saake and A.Sernadas (eds), *Proc. of the IS-CORE Workshop'91*, Informatik-Berichte 91-03, Tech. Univ. Braunschweig, 1991, 1-24

[Fiadeiro *et al* 91]
J.Fiadeiro, C.Sernadas, T.Maibaum and G.Saake, "Proof-theoretic Semantics of Object-oriented Specification Constructs", in R.Meersman, W.Kent and S.Khosla (eds), *Object Oriented Databases: Analysis, Design and Construction*, North Holland, 1991, 243-284

[Fishman *et al* 87]
D.Fishman *et al*, "IRIS: An Object-Oriented Database Management System", *ACM Trans. Off. Inf. Sys. 5*, 1987

[Goldberg and Robson 83]
A.Goldberg and D.Robson, *Smalltalk 80: The Language and its Implementation*, Addison-Wesley, New York, 1983

[Goguen 91]
J.Goguen, "Types as Theories", *Proc. Conf. on Topology and Category Theory in Computer Science*, Oxford University Press, 1991, 357-390

[Heitz 88]
M.Heitz, "HOOD: A Hierarchical Object-Oriented Design Method", *Proc. of the 3rd German Ada Users Congress*, Munich, 1988, 12-1—12-9

[Hoare 85]
C. A. R.Hoare, *Communicating Sequential Processes*, Prentice-Hall, 1985

[Jonsson 85]
B.Jonsson, "A Model and Proof System for Asynchronous Networks", in *Proc. of the 4th Annual ACM Symposium on Principles on Distributed Computing*, Minaki, Canada, 1985, 49-58

[Jonsson 89]
B.Jonsson, "A fully abstract trace model for dataflow networks", in *Proc. of the 16th Annual ACM Symposium on Principles of Programming Languages*, 1989, 155-165

[Jungclaus *et al* 91]
R.Jungclaus, G.Saake, T.Hartmann and C.Sernadas, *Object-Oriented Specification of Information Systems: The TROLL Language*, Informatik-Berichte, Tech. Univ. Braunschweig, 1991, *to appear*

[Kim *et al* 88]
W.Kim *et al*, "Features of the ORION Object-Oriented DBMS", in W.Kim and E.H.Lochovsky (eds), *Object-Oriented Concepts, Databases, and Applications*, Addison-Wesley, 1988

[Kerth 88]
N.Kerth, *MOOD: A Methodology for Structured Object-Oriented Design*, Tutorial presented at OOPSALA'88, San Diego, 1988

[Lukaszewicz 88]
W.Lukaszewicz, *Non-Monotonic Reasoning*, Ellis Horwood, 1990

[Meyer 88]
B.Meyer, *Object-Oriented Software Construction*, Prentice-Hall, Englewood Cliffs, 1988

[E.Seidewitz and M.Strak 86]
E.Seidewitz and M.Stark, *General Object-Oriented Software Development*, Document No. SEL-86-002, NASA Goddard Space Flight Center, Greenbelt, Maryland, 1986

[SernadasA *et al* 89]
A.Sernadas, J.Fiadeiro, C.Sernadas and H.-D.-Ehrich, "Basic Building Blocks of Information Systems", in E.Falkenberg and P.Lindgreen (eds), *Information System Concepts: An In-depth Analysis*, North-Holland, 1989, 225-246

[SernadasA and Ehrich 90]
A.Sernadas and H.-D.Ehrich, "What is an Object, After All", in R.Meersman, W.Kent and S.Khosla (eds), *Object-oriented Databases: Analysis, Design and Construction*, North-Holland, 1991, 39-69

[SernadasA *et al* 90]
A.Sernadas, H.-D.Ehrich and J.-F.Costa, "From Processes to Objects", *The INESC Journal of Research and Development* 1(1), 1990, 7-27

[SernadasC *et al* 91]
C.Sernadas, P.Resende, P.Gouveia and A.Sernadas, "In-the-large Object-oriented Design of Information Systems", in F.vanAssche, B.Moulin and C.Rolland (eds), *Object-Oriented Approach in Information Systems*, North-Holland, 1991, 209-232

[SernadasC *et al* 92]
C.Sernadas, P.Gouveia, J.Gouveia, A.Sernadas and P.Resende, "The Reification Dimension in Object-oriented Data Base Design", in D.Harper and M.Norrie (eds), *Specification of Data Base Systems*, Springer-Verlag, 1992, 275-299

[Stroustrup 86]
B.Stroustrup, *The C++ Programming Language*, Addison-Wesley, Reading, Mass., 1986

[Wegner 89]
P.Wegner, "Learning the language", in *Byte* 14, 1989, 245-253

[Winskel 84]
G.Winskel, "Synchronization trees", in *Theoretical Computer Science* 34, 1984

[Winskel 89]
G.Winskel, "An Introduction to Event Structures", in J.W. deBakker, W.P.de Roever and G.Rozenberg (eds), *Linear Time, Branching Time and Partial Order in Logics and Models for Concurrency*, LNCS 354, Springer-Verlag, 1989, 364-397

Appendix — Required Concepts of Category Theory

Concrete category. A *concrete category* over a base category B is a pair <A,U>, where A is a category and $U:A \to B$ is a faithful functor. Since faithful functors are injective on hom-sets, we assume that $hom_A(a_1,a_2)$ is a subset of $hom_B(U(a_1),U(a_2))$ for each pair $\langle a_1,a_2\rangle$ of A-objects. This convention allows one to express the property that "for A-objects a_1 and a_2 and a B-morphism $f:U(a_1) \to U(a_2)$ there exists a unique A-morphism $a_1 \to a_2$ such that $U(a_1 \to a_2)=f$" by stating "$f:U(a_1) \to U(a_2)$ is an A-morphism".

Amnestic Concrete Categories. Let <A,U> be a concrete category over B. (a) The *fibre* of an B-object X is the preordered class consisting of all A-objects Y with U(Y)=X ordered by: $Y \le Y'$ iff $id_X:UY \to UY'$ is an A-morphism. (b) A-objects Y and Y' are said *equivalent* provided that $Y \le Y'$ and $Y' \le Y$. (c) <A,U> is said to *amnestic* provided that its fibres are partially ordered classes, ie, no two different A-objects are equivalent.

Lifting of limits and colimits. Let <A,U> be a concrete category over B. A functor $U:A \to B$ is said to *lift limits* (uniquely) provided that for every diagram $D:I \to A$ and every limit L of $U \circ D$ there exists a (unique) limit L' of D with U(L')=L. Dual notion: *lift colimits* (uniquely). For concrete categories <A,U> over B the following conditions are equivalent: (a) U lifts limits uniquely and (b) U lifts limits and <A,U> is amnestic.

Sources and sinks. A *source* in a category A is a pair $\langle X,\{f_i\}_{i\in I}\rangle$, usually denoted by $(f_i:X \to X_i)_{i\in I}$, consisting of an object X and a family of morphisms $f_i:X \to X_i$ with domain X, indexed by some class I. A source $(f_i:X \to X_i)_{i\in I}$ in a concrete category <A,U> over B is called *initial* provided that a B-morphism $f:UY \to UX$ is a A-morphism whenever each composite $f_i \circ f:UY \to UX_i$ is a A-morphism. Let <A,U> be a concrete category over B. A limit L of a diagram $D:I \to C$ is called a *concrete limit* of D in <A,U> provided that it is preserved by U. A concrete limit can be constructed in two steps: first, form the limit of the underlying diagram in the base category, and, second, provide an initial lift of this underlying limit. In other words, if <A,U> is a concrete category over B and $D:I \to A$ is a diagram then $L=(f_i:L\to D_i)_{i\in \mathrm{obj}(I)}$ is a concrete limit in $\langle A,U\rangle$ iff $U(L)$ is a limit of $U \circ D$ and L is an initial source in $\langle A,U\rangle$.

The concept dual to that of *source* is called *sink*. Dual concepts of *concrete colimit* and *final sink* are left to the reader. A concrete colimit can be also constructed in two steps: first, form the colimit of the underlying diagram in the base category, and, second, provide a final lift of this underlying colimit.