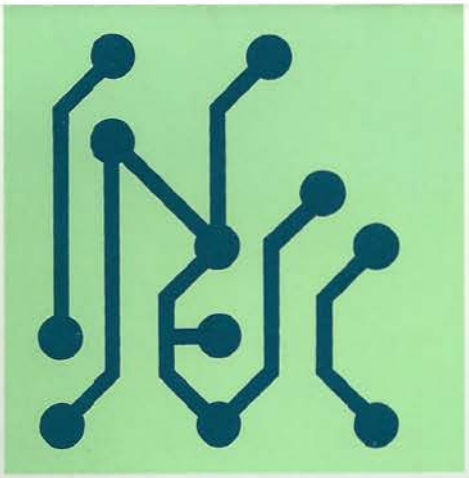
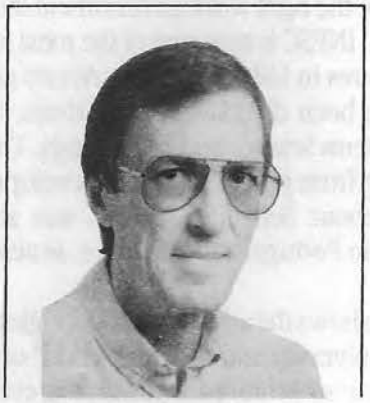


the INESC JOURNAL OF RESEARCH & DEVELOPMENT

LISBON • JAN/JUN-90

VOL.1 N.º 1

Amilcar Sernadas was born in 1952 in Angola. He graduated in Electrical Engineering at the Lisbon Institute of Technology in 1975 and obtained his PhD in Computer Science at the University of London in 1980. Since 1986 he is professor for Theoretical Computer Science at the Department of Mathematics of the Lisbon Institute of Technology (Instituto Superior Técnico) and leader of the Lisbon Computer Science Group at INESC.



Hans-Dieter Ehrich was born in 1943 in Schleswig. He graduated in Mathematics at the University of Kiel. Between 1967 and 1974 he has worked as scientific assistant at the Technical University of Hannover and at the University of Kiel. He has become professor for Theoretical Computer Science in 1974 at the University of Dortmund, and since 1982 professor for Databases and Information Systems at the Technical University of Braunschweig.

José-Félix Costa was born in 1959 in Lisbon. He received the Physics degree in 1982 from the Classical University of Lisbon. Since 1987 he has become scientific assistant at the Department of Mathematics of the Lisbon Institute of Technology (Instituto Superior Técnico) and a researcher of the Lisbon Computer Science Group at INESC. He received early in this year his MSc degree in Mathematics.

From processes to objects

A. Sernadas, H.-D. Ehrich** and J.-F. Costa**

Abstract

A mathematical model for objects is given as an extension of a suitable (interleaving) model for processes. Basically, an object is defined as a process endowed with trace-dependent attributes. Although the extension is largely independent of the chosen process model, a specific model for finite deterministic processes is adopted for illustration purposes: the simplest model supporting the distinction between active and passive processes, since such a distinction is shown to be important for objects. The joint behaviour of concurrent, interacting objects is described categorially. Several basic mechanisms for interaction are considered, including event sharing and event calling.

1 — Introduction

Object-oriented programming began as early as 1967 with the simulation language SIMULA [4], but its popularization is usually attributed to the advent of Smalltalk [11]. Object-orientation has been proposed as a computation paradigm by itself [16, 15]. However, after all these years it still lacks an effective mathematical foundation. We might even argue that the very notion of object is yet to be agreed upon. But many of its essential aspects have been «frozen» [17, 1, 5, 20], to the point where it seems to have become feasible to start working towards a mathematical model for objects.

In our previous work [19, 9, 8, 7, 6] we contributed to such a model for objects, object types, aggregation of concurrent, interacting objects and reification of objects with objects, capitalizing in some analogies from algebraic data type theory. Indeed, according to the object-oriented view, a stack is not a data type, but a single *object instance.* As an object, a stack has an internal *state* that can be changed by certain operations *(events)* and observed through slots *(attributes).*

On the other hand, as an object, the stack displays behaviour. That is, there is a *process* corresponding to its behaviour. Thus, basically, an object is a process endowed with trace-dependent attributes. Clearly, a mathematical model for objects must include a suitable model for processes.

Several convenient, sophisticated mathematical models for processes have been proposed, such as those reported in [2, 14], taking some or all of the basic three approaches: axiomatization, denotational semantics and operational semantics. Many relevant issues, like nondeterminism and the limitations of the interleaving models, have been clarified and are by now well understood. Other issues, like liveness or reification are yet under research. But, as a whole, the basic notion of a process, for instance according to the interleaving school of thought, is mathematically well founded. This nice state of affairs is reflected from a perspective outside of the field (where the authors place themselves) in Hennessy's textbook [13].

* Departamento de Matemática, Instituto Superior Técnico, 1096 LISBOA CODEX, PORTUGAL

** Abteilung Datenbanken, Tech Univ Braunschweig, Postfach 3329, 3300 Braunschweig, FRG

In this paper, stressing the strong relationship to the models for processes, we present an extended version of our previous mathematical model for objects, in order to model the assymmetrical form of interaction between two objects known as *event calling*. The proposed object model is largely independent of the chosen model for processes. But, for illustration purposes, a specific model for finite deterministic processes is adopted: the simplest model supporting the distinction between active and passive processes, since such a distinction is shown to be important for objects. This rather simplistic process model is sufficiently rich to illustrate the construction of the object model. The paper is dedicated to object communication. Other important issues in object-orientation (like typing and inheritance) are completely ignored.

As in [9, 8, 6] we make moderate use of a few category-theoretic notions. The reader may find all the necessary notions in the first chapters of Goldblat's textbook [12].

One of the most significant achievements of the proposed model for objects is a categorial construction of the aggregation of concurrent, interacting objects *(ibidem)*. Several basic mechanisms for interaction are considered, but emphasis is put herein in the event calling form of interaction that was not considered before. As a side effect, we obtain a categorial characterization of the parallel (interleaving) operation on processes, in the absence or in the presence of either symmetrical or assymmetrical communication.

In section 2, the concept of object is informally reviewed and related to the underlying notion of process. In order to provide the setting for the envisaged mathematical model for objects, a simple model for finite deterministic processes is given in section 3 that allows the distinction between active and passive processes. In section 4, the definition of object is given, adopting the process model of the previous section. In section 5, the parallel composition of processes is presented as a colimit in a suitable category, including the basic forms of symmetrical object interaction, as well as the traditional, assymetrical event calling mechanism. In section 6, the aggregation of objects is obtained by enriching parallel composition with the attributes of the objects.

2 — Objects *VS* processes

A computer system, when observed as a whole, is a symbolic *machine* that is able to perceive, manipulate, store, produce and transmit information. As such, the computer system is composed of two basic kinds of parts. On one hand, we have the *storage components* like files, records, databases and, of course, working areas in central memory. These storage components are able to keep lexical things like integers, names and so on, in general known as *data*. On the other hand, we have the *process components* like 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 the parts of the computer system. Forgetting data for the moment, both the storage and the process components have a distinct temporal and spatial existence. Any of them is created and evolves throughout time (ie, its state changes), possibly moving from one place to another, until it is finally destroyed (if it ever comes to happen). Any of them is able to retain data and is able to replace the data it is storing. Any of them can either be very persistent (with a long life) or transient (with a short life).

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

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

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

Hence, the basic building block of computer systems is the object, presented here as a process plus trace-dependent attributes. Moreover, objects can be put together through aggregation. The resulting joint behaviour corresponds to the parallel composition of the component processes. Thus, aggregation appears as the extension of parallel composition to attributes.

3 — A Simple model of active processes: fLC

The envisaged finite life-cycle model ***(fLC)*** of deterministic processes is obtained from the traditional finite traces model ***(PS)*** by dropping the prefix-closed condition. Thus, according to ***fLC***, any finite set of traces is a process. For instance, wrt the event alphabet $E=\{e_1,e_2\}$, the set $\{e_1e_1,e_1e_2,e_2e_1e_2\}$ is a ***fLC***-process, but not a ***PS***-process since it is not prefix-closed.

Clearly, every ***PS***-process is also a ***fLC***-process. Such a process is considered to be *passive*, since it is always ready to do «nothing». Those ***fLC***-processes that are not ***PS***-processes display some activity. Indeed, we say that the process $\{e_1e_1,e_1e_2,e_2e_1e_2\}$ above is *active* in the sense that, for instance, after e_2e_1 it is «willing» to make event e_2 happen.

Given any ***fLC***-process S we can calculate the corresponding passive process having the same trace language *pc(S)*: the least prefix-closed set of traces containing *S*. For instance, $pc(\{e_1e_1,e_1e_2,e_2e_1e_2\})=\{\varepsilon,e_1,e_1e_1,e_1e_2,e_2,e_2e_1,e_2e_1e_2\}$.

Two processes with the same prefix-closure are comparable wrt the activity degree: the less is the number of life-cycles, the greater is the activity. Thus, every process has at least the activity of its prefix-closure. This activity comparison should lead to a partial-order among the processes.

In order to generate the envisaged process domain, it is necessary to enrich the traditional language of finite, deterministic processes by distinguishing two forms of prefixing: active (!)

and passive (?). Informally, if p is a process term, both $e?p$ and $e!p$ are process terms. The former is traditionally written ep and corresponds to *passive prefixing*. The term $e!p$ corresponds to *active prefixing*. Naturally, we envisage to impose

$$[[e?p]] = \{\varepsilon\} \cup \{es: s \in [[p]]\}$$
$$[[e!p]] = \{es: s \in [[p]]\}$$

Thus, the process $\{e_1e_1, e_1e_2, e_2e_1e_2\}$ above is denoted, for example, by the process term $e_1!(e_1!+e_2!)+e_2!e_1!e_2!$. And its prefix-closure is denoted by $e_1?(e_1?+e_2?)+e_2?e_1?e_2?$. Between these two extremes we have several processes with the same trace language but different degrees of activity, such as $e_1?(e_1!+e_2!)+e_2!e_1?e_2?$.

The remainder of this section is dedicated to *fLC*-processes, presenting the envisaged process language, denotational partial-order semantics and inequational inference system. The corresponding operational semantics and suitable testing methodology is left out because they are too lengthy to include herein. We follow closely the algebraic characterization style adopted in [13] for finite, deterministic processes. Further details and proofs omitted herein can be found in [3]. Therein, the operational semantics, the testing methodology (able to distinguish between passive and active processes) and a full abstraction result (of the denotational semantics wrt the testing preorder) is also presented in detail.

Language of process terms

Given a countable alphabet E of events, the language corresponds to the **term algebra** T_Σ over the **signature** $\Sigma=\Sigma^0\cup\Sigma^1\cup\Sigma^2$, where $\Sigma^0=\{\textbf{\textit{nil}}\}$, $\Sigma^1=\{e?: e\in E\}\cup\{e!: e\in E\}$ *(prefixing)*, and $\Sigma^2=\{+\}$ *(choice)*. Each Σ^j, for $j=0..2$ is the set of the j-ary operation symbols. As usual, we write $e?p$ and $e!p$ instead of $e?(p)$ and $e!(p)$, respectively, as well as $e?$ and $e!$ instead of $e?(\textbf{\textit{nil}})$ and $e!(\textbf{\textit{nil}})$. Moreover, we assume that prefixing has priority over choice. Whenever necessary, given a set X (of variables), we shall also use the algebra $T_\Sigma(X)$ of terms over the signature Σ enriched with the elements of X as unary symbols.

Partial-order denotational semantics

We impose a *po* denotational semantics for the process terms by choosing a Σ-*po* algebra $A=<|A|,\leq_A,\Sigma_A>$, where $|A|$ is the carrier set (its elements are called A-processes), $\leq_A$ is a partial-order over $|A|$, and Σ_A provides for each n-ary symbol f of Σ a n-ary function f_A within $|A|$. Then, the denotation of a process term p is taken to be $i_A(p)$, where i_A is the unique homomorphism from T_Σ into A (recall that T_Σ endowed with the *po* identity is initial among the Σ-*po* algebras). Whenever the target algebra A is clear from the context, we write $i(p)$ and preferably $[[p]]$ instead of $i_A(p)$.

As expected, wrt such a model A, two process terms p and p' are said to be **equal** iff they denote the same A-process: $i_A(p)=i_A(p')$. Moreover, p is said to be **less active** then p', wrt A, iff $i_A(p)\leq_A i_A(p')$. That is, $\leq_A$ partially orders the A-processes according to their activity.

The envisaged ***fLC*** algebra is easily established:

$$\boldsymbol{fLC} = <|\boldsymbol{fLC}|, \leq_{fLC}, \Sigma_{fLC}>$$

where

$$|\boldsymbol{fLC}| = \{S:\ S \subset E^* \wedge 0<\#S<\#\omega\}$$

$$\leq_{fLC} = \{<S,S'>:\ pc(S)=pc(S') \wedge\ S' \subset S\}$$

$$\Sigma_{fLC} = \{\boldsymbol{nil}_{fLC},\ e?_{fLC} \text{ for } e \in E,\ e!_{fLC} \text{ for } e \in E,\ +_{fLC}\}$$

$$\boldsymbol{nil}_{fLC} : \rightarrow |\boldsymbol{fLC}| \qquad \boldsymbol{nil}_{fLC} = \{\varepsilon\}$$

$$e?_{fLC} : |\boldsymbol{fLC}| \rightarrow |\boldsymbol{fLC}| \qquad e?_{fLC}(S) = \{\varepsilon\} \cup \{es:\ s \in S\}$$

$$e!_{fLC} : |\boldsymbol{fLC}| \rightarrow |\boldsymbol{fLC}| \qquad e!_{fLC}(S) = \{es:\ s \in S\}$$

$$+_{fLC} : |\boldsymbol{fLC}|\ |\boldsymbol{fLC}| \rightarrow |\boldsymbol{fLC}| \qquad +_{fLC}(S,S') = S \cup S'$$

It is straightforward to prove the following result: ***PROP1:*** **The Σ-*po* algebra *fLC* is generated.** That is, for any $S \in |\boldsymbol{fLC}|$ there is a $p \in T_\Sigma$ such that $[[p]]=S$.

Inequational inference system

We now want to give a syntactical characterization of the ***fLC*** model through a suitably correct and complete inequational inference system. But before we need to recall a few concepts. As usual, an **inequation** with variables in X is of the form $p \leq p'$, where p and p' are terms of $T_\Sigma(X)$. Given a set In of inequations (said to be the proper axioms), we establish the least set of inequations $dc(In)$ containing In as well as $\cup_X\{p \leq p : p \in T_\Sigma(X)\}$ (**reflexivity**), and closed for **transitivity**, **substitution** and **instantiation**. This set is called the **derivation-closure** of In and contains all the theorems we can derive from In. Thus, we write $In \vdash p \leq p'$ iff $p \leq p' \in dc(In)$.

It is worthwhile to recall the substitution and instantiation rules, respectively:

$$p_1 \leq p_1',\ \ldots,\ p_k \leq p_k' \ \vdash f(p_1,\ldots,p_k) \leq f(p_1',\ldots,p_k')$$

$$p \leq p' \ \vdash p\rho \leq p'\rho \text{ for every variable substitution } \rho$$

Within this inequational inference system it is possible to use **equations as abbreviations** of the corresponding inequations (in both directions). That is, $p = p'$ is an abbreviation of $p \leq p'$ and $p' \leq p$.

The idea is to identify a set In_{fLC} of proper axioms such that $p \leq p'$ is a theorem iff $[[p]] <_{fLC} [[p']]$. It is not difficult to arrive at the following set:

$$In_{fLC} = \{A1, A2, A3\} \cup \{A4_e:\ e \in E\} \cup \{A5_e:\ e \in E\} \cup \{A6_e:\ e \in E\}$$

where (adopting the variables x, y and z)

$A1$ $\quad x+x = x$

$A2$ $\quad x+y = y+x$

$A3$ $\quad x+(y+z)=(x+y)+z$

$A4_e$ $\quad e!(x+y)=e!x+e!y$

$A5_e$ $\quad e!x+\mathbf{nil}=e?x$

$A6_e$ $\quad e?x\leq e!x$

This set is close to the one given in [13] for ***PS***. Naturally, the axiom $x+\mathbf{nil}=x$ had to be replaced (by $A5_e$) and the activity axiom schema ($A6_e$) had to be added. Note also that distributivity of passive prefixing over choice can be derived from the axioms above.

The correctness of $dc(In_{fLC})$ is easily verified. ***PROP2:*** **For every $p,p'\in T_\Sigma$, $In_{fLC}\vdash p\leq p'$ implies $[[p]]\leq_{fLC}[[p']]$.**

Moreover, the inference system is also complete. ***PROP3:*** **For every $p,p'\in T_\Sigma$, $[[p]]\leq_{fLC}[[p']]$ implies $In_{fLC}\vdash p\leq p'$.** This is a non trivial result. We only outline the proof herein. For details see [3].

First we identify in T_Σ the subset C_Σ of **canonical terms**. Assume that each sequence $s\in E^*$ is mapped to a process term $\mu(s)$ as follows: $\mu(\varepsilon)$ is the process term ***nil***; for $n>0$, $\mu(e_1...e_n)$ is the process term $e_1!...e_n!$. Then, a process term is said to be canonical iff it is of the form $+_{s\in S}\mu(s)$, for some non-empty, finite subset S of E^*. Clearly, $+_{s\in\{r\}}\mu(s)$ degenerates into $\mu(r)$. The process term ***nil*** is canonical because it has the form $+_{s\in\{\varepsilon\}}\mu(s)$. So is, for instance, $e_1!+e_1!e_2!$. But not $e_1!e_2?$.

The following lemma can easily be established by induction on the structure of the process term: **For every $p\in T_\Sigma$, $In_{fLC}\vdash p=+_{s\in[[p]]}\mu(s)$.** That is, any term is reducible to a canonical term, the equality being derivable from In_{fLC}.

Hence, it remains to prove the completeness for canonical terms: **For every $p,p'\in C_\Sigma$, $[[p]]\leq_{fLC}[[p']]$ implies $In_{fLC}\vdash p\leq p'$.** The proof is made by induction on the cardinality of $[[p']]$, using the following auxilliary result: **$In_{fLC}\vdash \mu(r)+\mu(s)\leq\mu(r)$ provided that $s\in pc([r])$.**

As a corollary of propositions 1, 2 and 3 we can obtain the initiallity of the Σ-*po* algebra ***fLC*** among all algebras satisfying In_{fLC}.

In the sequel, it will be important to make explicit the alphabet of events. Thus we denote by ***fLC***(E) the set of ***fLC***-processes over the alphabet E. Moreover, $|fLC|$ will denote the set of all pairs of the form (E,S) where E is an alphabet and S is a ***fLC***-process over that alphabet E (that is $S\in$***fLC***(E)). As we shall see in section 5, $|fLC|$ is the class of objects of the category of ***fLC***-processes.

4 — A simple model of objects: fOB

Given a process over an alphabet of events E, we obtain an object by endowing the process with an alphabet of attributes plus an attribute valuation map providing their values after each trace of events. In the sequel we adopt the ***fLC*** model of processes, although it should be clear that another model can be used instead of ***fLC***, as long as it distinguishes between active and passive processes.

Thus, herein we define an **object** ob as a triple composed of an alphabet of events, a ***fLC***-process over that alphabet, and an observation structure over that alphabet:

$$ob = (E, S, V)$$

where:

E is a countable set (the **alphabet of events**);

S is a ***fLC***-process over E (the **behaviour**), that is: $S \in \mathit{fLC}(E)$ and $(E,S) \in |\mathit{fLC}|$;

$V = (A,\alpha)$ is the **attribute observation structure** over E, that is:

A is a set (the **alphabet of attributes**);

α: $E^* \rightarrow Obs$ is the **primitive attribute valuation map**, where:

$$Obs \subset 2^{A \otimes \text{integer}}$$

such that $\alpha(\varepsilon) = \varnothing$.

The map α returns a collection of attribute-value pairs for each finite sequence of events (even if the sequence is not in $pc(S)$). Given such a sequence s of events, for each $a \in A$, if there is no integer i such that $(a,i) \in \alpha(s)$ we say that the attribute a is undefined after the sequence s. It may happen that no attribute is defined after some trace (whenever α returns the empty observation). It may also happen that an attribute is given more than one value (when α returns an observation containing more than one pair for that attribute). Thus, wrt attribute values this model allows for nondeterminism.

Naturally, we can equivalently look at α as a map from E^* into the set of functions from A into 2^{integer}. Then, $\alpha(s)(a)$ denotes the set of values of attribute a after the sequence s.

For the sake of simplicity we are considering that all attributes are integer valued, but the framework is easily extended to a many-sorted data universe.

Clearly, at a given «time», the **state** of the object is derived from the sequence of events that have already happened until then.

As an example of a passive object consider the following «stack»:

$$E_{stack} = \{new, drop, pop\} \cup \{push(i): i \in integer\}$$

$$A_{stack} = \{top, empty\}$$

Typically, the valuation map is specified using a suitable action logic language. Adopting the language proposed in [10], we write the following equations about α_{stack}, for every $i \in$ integer and $s \in E^*$:

[new]empty = 1
[push(i)]empty = 0
[push(i)→pop→s]empty = [s]empty
[push(i)]top = i
[push(i)→pop→s]top = [s]top

Please note that, for any state r, the term *[s]a* denotes the value(s) of attribute a after the sequence rs. Naturaly, for that state r, the term a denotes the value(s) of attribute a after the sequence r.

Wrt the behaviour of the stack there is not much to specify, since it is a passive object accepting almost any sequence of events. We only want to allow the happening of a pop event only if the stack is not empty (that is, only if empty = 0). Still using the action language above we write:

{empty = 0} pop

This form of safety constraint is very common: in general *{c}e* means that, for any state r, the event e is allowed only if condition c is true in that state.

No matter how we specify the behaviour, we should get a set of (finite) life-cycles:

$$S_{stack} = \{s \in E^*: \forall r \in E^* \; r.pop \in pc(s) \Rightarrow \alpha(r)(empty)=\{0\}\}$$

This example shows why we wanted to consider the map α defined over E^* and not only over the prefixes of admissible life-cycles.

As another example consider the active object «user» willing to make a stack with the first three positive integer powers of 2:

$$E_{user} = \{begin, end, new\} \cup \{push(i): i \in integer\}$$

$$A_{user} = \varnothing$$

In this case, the map α is constantly equal to $\varnothing$. On the other hand, the intended behaviour is easily specified in the same action language as follows:

begin→new→push(2)→push(4)→push(8)→end

That is, using the equivalent process term of T_Σ, we have:

$$S_{user} = [[\text{new push(2) push(4) push(8)}]]$$

Obviously, we would like to combine these two objects (user and stack) in order to obtain the composite object displaying their joint behaviour (taking into account the shared events). That we shall do in section 6. But before we need to introduce the mechanisms for combining processes.

In the sequel, the set of all objects as defined above will be denoted by $|fOB|$: the class of objects of the category fOB to be introduced in section 6.

5 — Putting processes together: parallel composition

The basic form of combining processes is parallel composition. The parallel composition of two independent processes can be modeled through the free interleaving of the components life-cycles. When the components interact, the resulting behaviour can be more difficult to describe, depending on the form of interaction. We shall consider only two forms: **event sharing** and **event calling**.

These two basic forms of process interaction have been found to be sufficient in practice (at least wrt object behaviour). The former corresponds to having a non-empty intersection of the event alphabets: the joint life-cycles still appear as interleavings of the components life-cycles, but «gluing together» the common events. As an example consider the sharing of events between the «user» and the «stack» objects in the previous section. This is a symmetrical form of interaction.

Although effective in some cases, (symmetrical) event sharing is not convenient when we do need some sort of assymmetrical interaction. That would be the case if we wanted two different «use» objects to be able to push into the «stack» without making a push made by $user_1$ also a push in the life of $user_2$ and vice-versa!

A partial order >> is assumed to exist over the alphabet of events. This partial order is necessary for the event calling mechanism. For that reason, the partial order >> is known as the **calling relation**. The intuition is the following: when an event e «happens», all events called by e also «happen». This will be discussed and illustrated at the end of section 6.

The set of finite life-cycle processes over an alphabet E of events endowed with a calling partial order will be denoted by $fLC_{>>}(E)$. Note that $fLC(E) \subset fLC_{>>}(E)$ in the sense that we can look at the original **fLC**-processes as $fLC_{>>}$-processes endowed with the trivial identity partial order. In the sequel, $|fLC_{>>}|$ will denote the set of all pairs of the form $[(E,>>),S]$ where $(E,>>)$ is an alphabet endowed with a calling partial order and S is a fLC-process over that alphabet E [that is $S \in fLC(E)$].

As remarked before, throughout the paper we retain the so called **interleaving model of parallelism**. Our results have yet to be extended to a non-interleaving model of true concurrency.

Returning to the parallel composition of two independent processes (the case of n processes does not raise any difficulty), let us consider the following $\mathbf{fLC}_{>>}$-processes:

$$pr_1 = [(E_1,>>_1),S_1]$$
$$pr_2 = [(E_2,>>_2),S_2]$$

The envisaged parallel composition (assuming no interaction) will be:

$$pr_1 \,||\, pr_2 = [(E_1 \oplus E_2, >>_1 \oplus >>_2), S_1 \,||\, S_2]$$

where:

$$>>_1 \oplus >>_2 = \{(e_1,e_2) \in (E_1 \oplus E_2) \otimes (E_1 \oplus E_2):\ (e_1,e_2)\downarrow E_1 \in >>_1 \ \vee\ (e_1,e_2)\downarrow E_2 \in >>_2\}$$
$$S_1 \,||\, S_2 = \{s \in (E_1 \oplus E_2)^*:\ s\downarrow E_1 \in S_1 \ \wedge\ s\downarrow E_2 \in S_2\}$$

where $\oplus$ stands for the disjoint union, $\otimes$ for the cartesian product, and $\downarrow$ for projections reversing the alphabet injections. The set of joint life-cycles is the envisaged free interleaving of the life-cycles in S_1 with those of S_2 (after renaming via the alphabet injections). The resulting calling relation is the least po in $E_1 \oplus E_2$ that reflects the calling relations of both components (also after renaming).

Process morphisms

The reasoning above suggests that a suitable notion of process morphism (enriching the notion of set-theoretical event map) could lead to obtaining $pr_1 \,||\, pr_2$ as «the» coproduct of pr_1 and pr_2 (unique up to isomorphism). Such a notion should make isomorphic only processes with identical alphabets and life-cycle sets (up to injective renaming of events). The solution is trivial if we consider only injective event maps. But, as we shall see later on for the case of interacting processes, in general we also need non-injective maps. Indeed we shall need coequalizers since, for instance, parallel composition appears as the pushout of the two processes over the shared events. Adapting from [6], we arrive at the following (most general) notion of process morphism.

A **process morphism** $h: pr_1 \to pr_2$ is an **event map** $h: E_1 \to E_2$ such that:

(1) The **calling inheritance condition** (CIC) is satisfied:

$$\forall e,e' \in E_1:\ \ e>>_1 e' \ \Rightarrow\ h(e)>>_2 h(e')$$

(2) The **life-cycle inheritance condition** (LIC) is satisfied:

$$\forall s_2 \in S_2\ \exists s_1 \in S_1:\ \ s_1 \in h^{\leftarrow}(s_2)$$

where

$$h^{\leftarrow}(e_1 \ldots e_n) = h^{\leftarrow}(e_1) \ldots h^{\leftarrow}(e_n)$$

and

$$h^{\leftarrow}(e) = \mathrm{PERM}(\{e' : h(e')=e\})$$

where, as expected,
PERM(Z) is the set of all permutation sequences of elements of set Z.

Requirement (1) means that the process morphism is a *po* morphism between the partially ordered alphabets of events. Requirement (2) is easily understood: every life-cycle of pr_2 should correspond to a life-cycle of pr_1 when events are mapped back.

Category of processes

With these morphisms we establish a **category of processes** (that we shall denote by $\mathbf{fLC}_{>>}$) whose objects are the $\mathbf{fLC}_{>>}$-processes and whose arrows are the morphisms we just introduced. Indeed, it is straightforward to verify that these process morphisms compose and that this composition is associative. Moreover, it is trivial to recognize the identity morphisms.

Let us examine the properties of this category vis a vis the parallel composition of processes without and with interaction. As we said, we hope to obtain the joint behaviour of two independent processes through the **coproduct** construction. And we also expect to obtain the joint behaviour of two processes sharing an event through the **pushout** construction. Within the framework of a closely related category of objects, these goals had already been achieved in [6]. The main novelty will be the construction of the joint behaviour of two processes in the presence of event calling.

When calculating colimits in $\mathbf{fLC}_{>>}$ we shall make extensive use of the corresponding constructions in cSET (the category of countable sets) and in cPO (the category of partial orders over countable sets). For instance, disjoint union is obtained via the coproduct construction in cSET. Recall that these categories are fully cocomplete: colimits exist for all the diagrams that we shall consider in this paper (and much more, but that is irrelevant). Comments on the cocompleteness of $\mathbf{fLC}_{>>}$ and $\mathbf{fOB}_{>>}$ (to be introduced in section 6) will be delayed until the concluding remarks.

Parallel composition of non-interacting processes

Consider again the two following $\mathbf{fLC}_{>>}$-processes:

$$pr_1 = [(E_1, >>_1), S_1]$$
$$pr_2 = [(E_2, >>_2), S_2]$$

Assuming no interaction, we would like to obtain their parallel composition $pr_1 || pr_2$ as the coproduct in $\mathbf{fLC}_{>>}$ (the process $pr_1 || pr_2$ in fig. 5.1 with the injections in_1 and in_2): for every pr

such that there are morphisms f_1 and f_2 respectively from pr_1 and pr_2 into pr, there should exist a unique morphism k: $pr_1 \,||\, pr_2 \rightarrow pr$ fulfilling $f_1 = in_1$; k and $f_2 = in_2$; k (that is, ensuring the commutativity of the diagram).

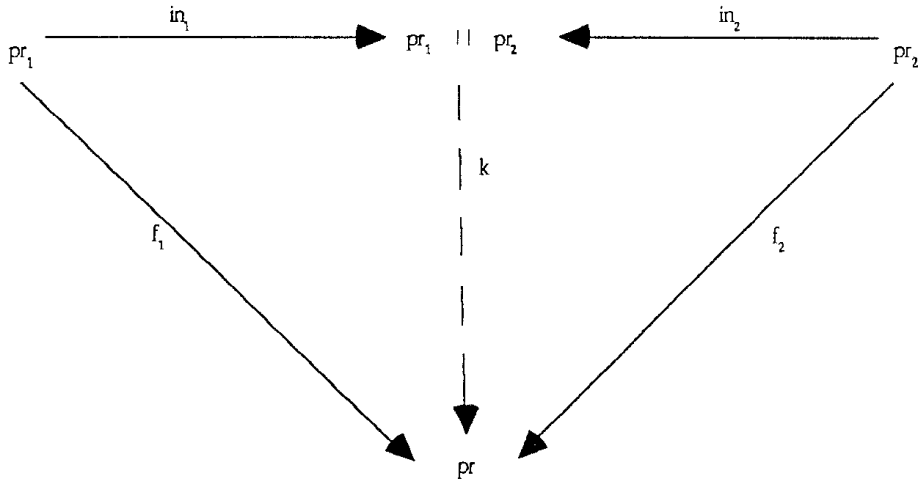


Fig. 5.1

This is rather easy to verify capitalizing on the coproduct constructions in cSET and cPO. Indeed, $E_1 \oplus E_2$ is the coproduct of the alphabets (taking in_1 and in_2 as the set injections from each of the alphabets into their disjoint union). Moreover, $(E_1 \oplus E_2, >>_1 \oplus >>_2)$ is the coproduct of the event partial orders (where $>>_1 \oplus >>_2$ is as described before: the «disjoint union» of the two partial orders). Thus, it remains to verify that the set morphisms (also *po* morphisms) in_1, in_2 and k do fulfil the life-cycle inheritance condition (LIC). In any case, the uniqueness of k is ensured already by the coproduct construction in cSET.

The fact that in_1 and in_2 fulfil the LIC is very simple to prove because they are injective maps. In such cases, the LIC simply requires that the projection on the map domain of each codomain life-cycle must be a domain life-cycle. But that was precisely our choice above for $S_1 \,||\, S_2$: $\{s \in (E_1 \oplus E_2)^* : s{\downarrow}E_1 \in S_1 \wedge s{\downarrow}E_2 \in S_2\}$.

Finally, recalling from the construction in cSET that k is $f_1 \oplus f_2$, it is simple again to verify the satisfaction of the LIC by k. Indeed, consider any $s' \in S$. We have to find at least one $s \in S$ belonging to $k^{\leftarrow}(s')$. Some s in the set $in_1(s_1) \,||\, in_2(s_2)$, where $s_1 \in S_1$ fulfils the LIC wrt f_1 [that is $s_1 \in f_1^{\leftarrow}(s')$] and $s_2 \in S_2$ does so for f_2, will do because that set is included in $S_1 \,||\, S_2$. Just pick the right interleaving sequence that reflects that in s'.

In conclusion, **binary coproducts exist in $\mathbb{LC}_{>>}$ and provide the disjoint parallel composition of the argument processes**: disjoint union of alphabets and calling relations, plus free interleaving of the life-cycles. This result is easily extended to finite and even to small coproducts, but that is beyond the scope of this paper.

Parallel composition of event sharing processes

Consider again two $\mathbf{fLC}_{>>}$-processes pr_1 and pr_2, but now assume that they are sharing a single event e. That is, the alphabet of their parallel composition $pr_1 \|_e pr_2$ should correspond to the set-theoretic union of the two alphabets, assuming that their intersection is $\{e\}$. Within cSET, this corresponds to a pushout construction. Thus, let us try it within our category $\mathbf{fLC}_{>>}$.

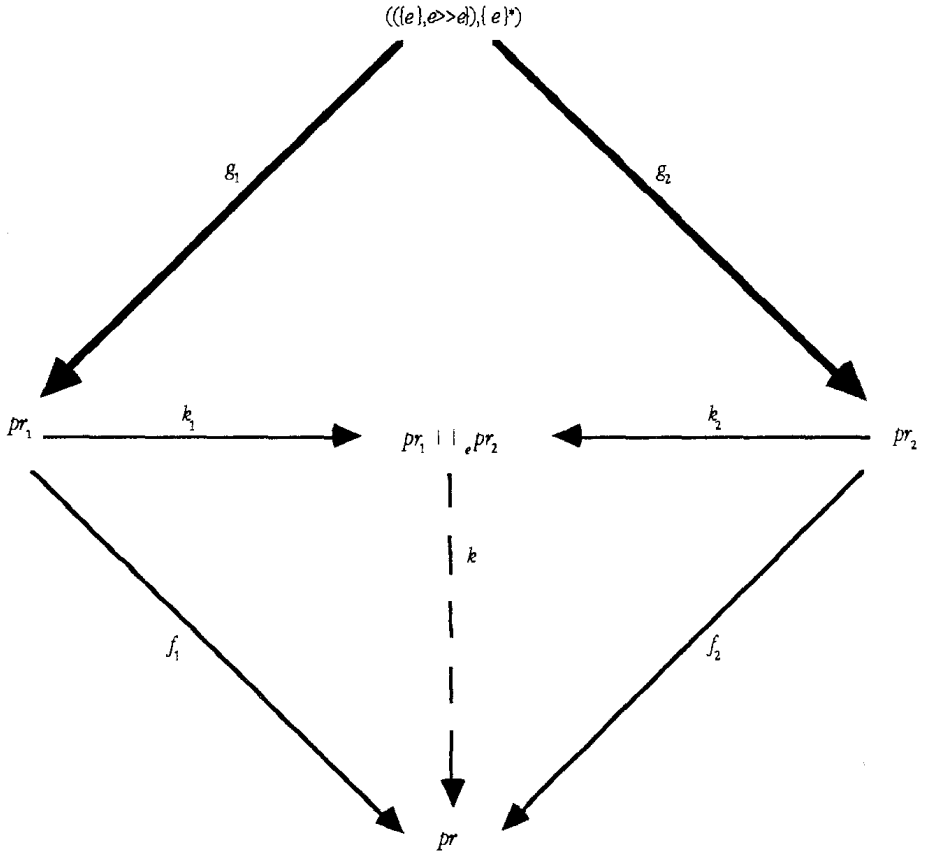


Fig. 5.2

Consider fig 5.2. Let us calculate the pushout process $pr_1 \|_e pr_2$ (that is, the colimit over the inverted V-diagram composed of arrows g_1 and g_2 and their common domain — the process $((\{e\},\{e>>e\}),\{e\}^*)$ at the top of the figure).

Note that the vertex of the inverted V is a process with a single event in the alphabet (the event e we want to share), the (unique) trivial partial order over it, and containing all possible sequences of that event as life-cycles. Thus, sharing the event e means sharing this entire process. Note that, in this way we ensure that the event maps g_1 and g_2 are indeed process morphisms.

The calculation in cSET is made as follows: first obtain the coproduct $E_1 \oplus E_2$ and then apply the coequalizer of g_1; in_1 and g_2; in_2. The result is (up to isomorphism):

$$E_1 \oplus_e E_2 = (E_1 \oplus E_2 - \{in_1(g_1(e)),\ in_2(g_2(e))\}) \oplus \{e\}$$

The calculation in cPO leads to (also up to isomorphism):

$$>>_1 \oplus_e >>_2 = tc\ ((\ >>_1 \oplus >>_2\ \cap (E_1 \oplus E_2 - \{in_1(g_1(e)),\ in_2(g_2(e))\})^2) \cup \{e>>e\ \})$$

where *tc* denotes the transitive closure operation.

Finally, wrt the life-cycles we obtain as desired

$$S_1 \oplus_e S_2 = \{s \in (E_1 \oplus_e E_2)^*:\ s{\downarrow}E_1 \in S_1 \wedge s{\downarrow}E_2 \in S_2\}$$

Only this last assertion needs verification, since the previous ones are known in cSET and cPO, respectively. The most direct way is similar to what we did for coproducts: it is straightforward to verify that the event maps k_1, k_2 and k do satisfy the LIC.

In conclusion, **pushouts over a single event exist in $\mathrm{fLC}_{>>}$ and provide the parallel composition of the argument processes**: «union» of alphabets, transitive closure of the «union» of the calling relations, and interleaving of the life-cycles «glued together» at the shared event.

Parallel composition of event calling processes

As we have just seen, colimits over some standard diagrams within $\mathrm{fLC}_{>>}$ model both the parallel composition of two independent processes and the parallel composition of two processes sharing a common event.

The question now arises: is it possible to find a simple diagram whose colimit is the complex object resulting from putting together two objects when one of them «calls» the other? Again, the answer is yes. The diagram is called an inverted W-diagram because of its characteristic shape (see fig. 5.3).

Note that the processes pr_1 and pr_2 do not share any event! The interaction goes through the middle vertex of the W: $((\{e_1,e_2\},\{e_1>>e_2, e_1>>e_1, e_2>>e_2\}),\{e_1,e_2\}^*)$. This process imposes the desired call $e_1>>e_2$.

The calculation of the colimit *pr* proceeds along the lines used for coproducts and pushouts: we determine the resulting alphabet and partial order within cSET and cPO respectively; then we determine the set of life-cyles. It is straightforward to obtain:

$$E_1 \oplus_{e1>>e2} E_2 = E_1 \oplus E_2$$

$$>>_1 \oplus_{e1>>e2} >>_2 = tc\ (\ >>_1 \oplus >>_2 \cup \{k_0(e_1)>>k_0(e_2)\}\ )$$

$$S_1\ ||_{e1>>e2}\ S_2 = S_1 || S_2$$

In conclusion, **colimits over a single event call ($e_1>>e_2$) exist in $\mathrm{fLC}_{>>}$**. Note that the call ($e_1>>e_2$) is only reflected in the partial ordering of the resulting colimit process. We delay until the next section further comments and illustration of this construct.

Note that mutual event calling (say $e_1>>e_2$ and $e_2>>e_1$) amounts to event sharing ($e_1=e_2$).

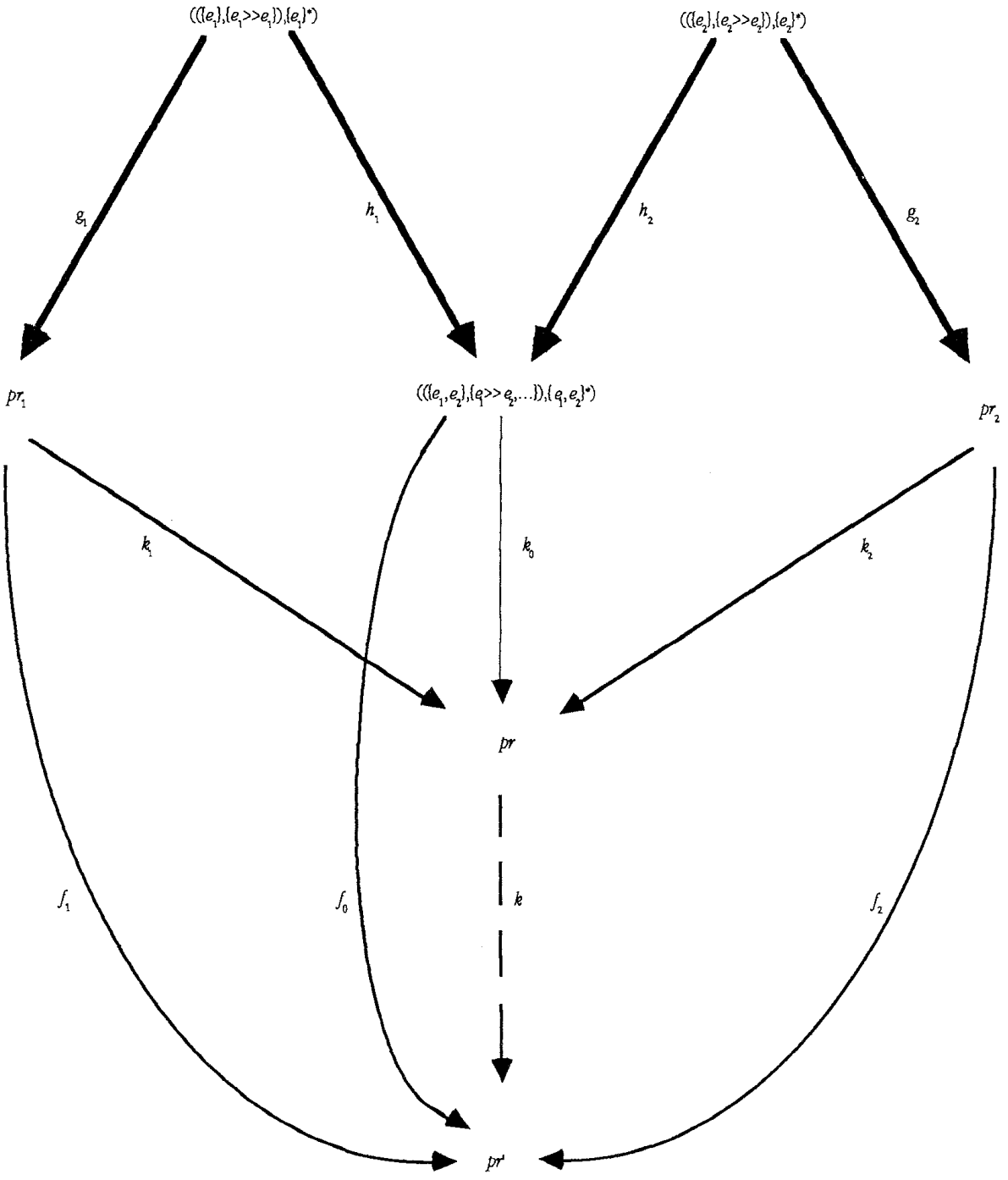

(([e1],{[e1>>e1]}),{[e1]*)
(([e2],{[e2>>e2]}),{[e2]*)
g1
h1
h2
g2
pr1
(([e1,e2],{[e1>>e2,...]}),{[e1,e2]*)
pr2
k1
k0
k2
pr
k
f1
f0
f2
pr'

Fig. 5.3

6 — Putting objects together: aggregation

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

Object interaction is traditionally classified into two categories: **memory sharing** and **communication**. According to the proposed model, the former corresponds to **attribute sharing** and the latter either to **event sharing** or to **event calling**. Attribute sharing and event sharing are special cases of **object sharing** (as originally recognised in [9]) which, naturally, also explains component sharing in general. Herein, we shall not discuss attribute sharing.

The idea is to enrich the concepts and constructs of the previous section endowing processes with attribute observation structures. But before we have to enrich the notion of object introduced in section 4 in order to introduce the calling relation. Accordingly, an object *ob* is now taken to be composed of a partially ordered event alphabet, a behaviour, and an attribute observation structure:

$$ob = [(E,>>),S,V]$$

The only difference is the inclusion of the partial order >> over E. Both S and V remain as before. However, as we shall explain in detail at the end of this section, the valuation map is now interpreted as a function of sequences of «clusters» of events (each «cluster» is composed by all the events called by a given event). Since the choice of the event uniquely determines the cluster we can retain E^* as the domain of the valuation map.

In the sequel, the set of all objects as just defined will be denoted by $|\mathbb{OB}_{>>}|$. We are now ready to introduce the notion of object morphism.

Object morphisms

An **object morphism** h: $ob_1 \rightarrow ob_2$ is a pair (h_E, h_A) composed of a **process morphism**

$$h_E: [(E_1,>>_1),S_1] \rightarrow [(E_2,>>_2),S_2]$$

and an **attribute map**

$$h_A: A_1 \rightarrow A_2$$

such that the **observation inheritance condition** (OIC) is satisfied:

$$\forall s_2 \in E_2^* \; \exists s_1 \in h_E^{\leftarrow}(s_2): \quad \alpha_1(s_1) = h_A^{\leftarrow}(\alpha_2(s_2))$$

where

$$h_A^{\leftarrow}(\rho) = \{(a,i): a \in A_1 \wedge (h_A(a),i) \in \rho\}$$

for any $\rho \in Obs_2$.

Comparing with the definition given in [6], the only difference is the addition of the CIC (implicit in the requirement that h_E be a process morphism) wrt the calling relations that were not considered therein.

The category of objects

With these morphisms we establish a **category of objects** (that we shall denote by $fOB_{>>}$) whose objects are the objects above using partially ordered alphabets of events and whose arrows are the morphisms we just introduced. Indeed, again, it is trivial to verify that these object morphisms compose and that this composition is associative. Moreover, it is also trivial to recognize the identity morphisms.

The full subcategory of the category OB introduced in [6] containing those objects only with finite life-cycles corresponds naturally to the full subcategory fOB of $fOB_{>>}$ containing those objects that are endowed with the trivial identity partial order.

We are now ready to characterize the aggregation of objects (without and with interaction) as colimits in this category. It will be a straightforward extension of the results of the previous section, taking into account the attributes and their valuation.

Aggregation of non-interacting objects

The coproduct construction presented in the previous section for processes is enriched with the disjoint union of attributes $A_1 \oplus A_2$ and the «sum» $\alpha_1+\alpha_2$ of the valuation maps, defined as follows for each $s \in (E_1 \oplus E_2)^*$:

$$(\alpha_1+\alpha_2)(s) = in_{1A}(\alpha_1(in_{1E}^{\leftarrow}(s)) \cup in_{2A}(\alpha_2(in_{2E}^{\leftarrow}(s))$$

Clearly, we are extending attribute maps to observations as expected:

$$h_A(\{(a,i),...\}) = \{(h_A(a),i),...\}$$

We have only to verify that the observation inheritance constraint is verified by the attribute maps used in the construction (refer to fig. 5.1 adapting it to objects and object morphisms): in_{1A}, in_{2A} and k_A. This is easily done.

Thus, **binary coproducts exist in $fOB_{>>}$ and provide the disjoint aggregation of the argument objects**: disjoint union of alphabet of events and disjoint parallel composition of behaviours, plus disjoint union of alphabets of attributes and addition of valuation maps. Again, this result is easily extended to finite and even to small coproducts.

As an example consider the coproduct of the objects stack and user introduced in section 4: stack||user. Note that we are extending the traditional notation for process parallel composition (||) to object aggregation, since they are so closely related. For instance, the alphabet of the resulting aggregation contains two distinct «pop» events, say stack.pop and user.pop in a notation that shows their different origins. This is not quite what we have in mind for putting together these two objects: instead we would like them to share (or to call) the stack events (eg pop's and push's), as will be analysed in the sequel.

Aggregation of event sharing objects

The pushout construction for processes has to be enriched with the disjoint union of attributes $A_1 \oplus A_2$ and the «sum» $\alpha_1+\alpha_2$ of the valuation maps (extending the definition given above for the sum of the valuation maps by replacing in_1 and in_2 by k_1 and k_2, respectively: see fig. 5.2). It is straightforward to verify that the resulting object is the envisaged pushout within $fOB_{>>}$. We denote this object by $ob_1 ||_e ob_2$.

Thus, **pushouts over a single event exist in $fOB_{>>}$ and provide the aggregation of the argument objects**: «union» of alphabets of events and pushout of behaviours, plus disjoint union of alphabets of attributes and sum of valuation maps.

These two constructions (coproduct and pushout) were already known within the setting of the category OB (see [6]). Let us now see how the partial order on events is used for explaining the event calling interaction mechanism between two objects.

But before consider again the objects stack and user. For instance, the aggregation $stack||_{new}user$ corresponds to putting together the stack and the user while sharing only the event new. Of course, this construction can be extended to sharing several events.

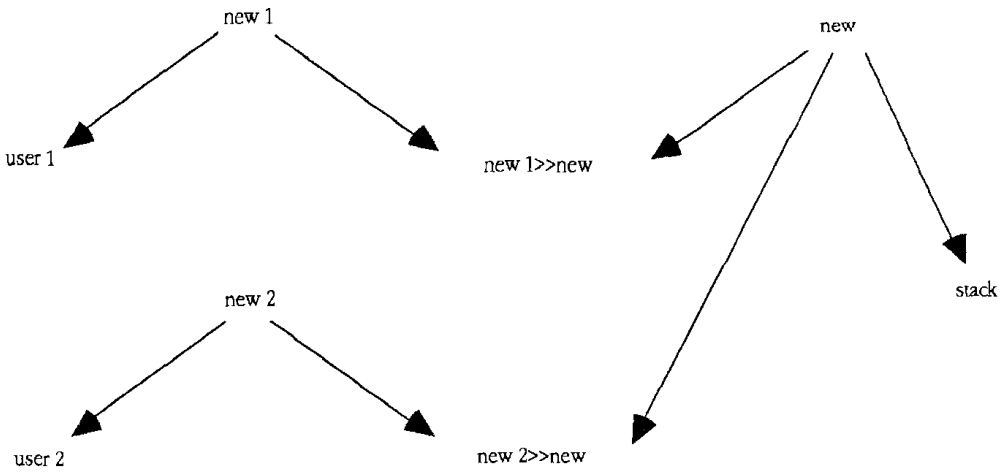
Aggregation of event calling objects

The aggregation of two objects ob_1 and ob_2 in the presence of single event calling (say $e_1>>e_2$) is also easily obtained enriching the W-colimit of the underlying processes with the disjoint union of atributes and the sum of valuation maps. We denote this colimit object by $ob_1 ||_{e1>>e2} ob_2$. Note that the call $(e_1>>e_2)$ is only reflected in the partial ordering of the resulting colimit object.

Clearly, **the *W*-colimits over a single event call $(e_1>>e_2)$ exist in $fOB_{>>}$**. Let us examine in detail the practical significance of such a construction.

Consider the following three objects: stack, user1 and user2. The stack is as introduced in section 4. The other two objects are isomorphic copies of the object user also introduced in that section. The main interest of the W-colimit over a call like new1>>new by objects user1 and stack is to make sure that every new1 event (of user1) implies a new event (of stack), but not vice-versa. Otherwhise, we would also identify new1 with new2, that is, the two user objects interacting via the stack would have their events synchronized!

With event calling we avoid the symmetry of event sharing. In this case (considering only the new events for the moment) what we want is the colimit of the following diagram:



Note that the two event callings (new1>>new and new2>>new) only affect the partial order of the events in the colimit object. But what we want is to make sure that whenever new1 happens the effects of new should take place (and similarly for new2). By «effects» we mean consequences both on attribute valuation and on further event interactions (for instance, if the event push calls or is identified to the event xyz of some other object).

With respect to any further event interaction, the fact that event calling is reflected in the partial order of the events of the colimit is enough. Indeed, transitivity and reflexivity guarantee that.

On the other hand, with respect to attribute valuation, things are a bit trickier. Indeed, after the event new of stack, the attribute empty takes the value 1 (recall the specification given in section 4). Thus we would like the same to happen in the colimit object whenever new1 happens (and the same for new2). Changing the partial order of events does not seem to do the job, since the valuation is given by the map α that was not affected by the calls according to our construction.

However, we must look at the valuation map α from a different point of view. Although we say that α maps sequences of events into observations as before, in the presence of calls the observation $\alpha(e_1...e_n)$ represents the observation after the sequence of «clusters» $e_1\wedge...e_n\wedge$, where each $e_i\wedge=\{e: e_i>>e\}$. Since each event uniquely determines its cluster, there was no need to make α a map from sequences of clusters into observations. Obviously, the same applies to the set S of life-cycles: each life-cycle is still a sequence of events, but where each of them stands for the corresponding cluster. That is, when an event «happens» we take that as the happening of all events in its cluster. Note that in the absence of calls every cluster is a single set and we return to the original view.

Thus, in the case at hand, in the colimit object over the diagram above, for instance, α(start1 new1) denotes the observation after the sequence start1$\wedge$ new1$\wedge$, that is, after the sequence of clusters {start1}{new1,new}. The same reasoning applies to new2.

Clearly, when calculating attribute observations in the presence of a calls, every valuation rule applies to clusters and not simply to events. For example, the rule

$$[new]empty = 1$$

applies also to every cluster containing new, namely the cluster of new1. But that is more an issue of the semantics of our object description language than an issue of the category of objects.

Again, as already pointed out for processes, mutual event calling by two objects amounts to event sharing.

Finally, note that we can easily extend the W-colimit construction to situations of multiple calling by the same object (eg user1 calls several events of stack) and/or of different calls of the same event by different objects (like in the diagram above).

7 — Concluding remarks

We presented a mathematical model for objects displaying finite, deterministic behaviour, either active or passive. Objects appear as processes endowed with attributes. Although we used a specific process model, it should be clear that the object concept can be built upon any other more sophisticated process model (eg allowing for nondeterministic and/or infinite processes). Naturally, this seems to be an interesting line of research. However, we stress the basic requirements that the candidate process model should fulfil: (1) ability to distinguish active from passive objects, since that distinction is essential to the object view of the computer system; (2) sufficient cocompleteness of the associated category of processes so that parallel compositions appear as colimits.

Actually, the category $\mathbb{fLC}_{>>}$ (as well as $\mathbb{fLC}$) of processes it is not fully cocomplete, not even countably cocomplete. Indeed, for instance, it is very simple to find multiple event sharing situations where the construction proposed in section 5 does not produce a colimit (that is, the process we obtain is not a colimit: there is no colimit). It seems that this is the case whenever the behaviours of the argument processes allow for the possibility of deadlock, but this issue needs further research.

Anyway, $\mathbb{fLC}_{>>}$ is sufficiently cocomplete in the sense that we can explain the different sorts of parallel composition (without or with interaction) as colimits. Moreover, that high degree of cocompleteness carries through to $\mathbb{fOB}_{>>}$ making possible to calculate the aggregation of a community of interacting objects as a colimit. The resulting object will be the colimit of the entire interaction diagram (in the absence of such deadlock situations).

It should be stressed that, although our main objective was the presentation of objects as enriched processes (with trace dependent attributes), along the way we presented a simple model for finite, deterministic processes dealing explicitly with their liveness, including a sound and complete inequational inference system. Within this context, the algebraic approach to the theory of processes presented in [13] was most helpful. For another process model also covering liveness but dealing with nondeterministic processes see [3].

Moreover, we also introduced the notion of process morphism and gave a categorial account of parallel compositon, following previous work along these lines but on the context of objects (see [9, 8] and, more recently, [6]). We are now actively applying these ideas to other process models, namely including nondeterminism.

ACKNOWLEDGEMENTS

The authors are indebted to their colleagues Francisco Dionísio (who brought forward the first example of a diagram without colimit), Cristina Sernadas and Gunter Saake for many useful discussions on several aspects of the problems addressed in the paper. This work was partially supported by the CEC through the ESPRIT-II BRA Nº 3023 (IS-CORE).

REFERENCES

[1] AMERICA, P. — «Object-Oriented Programming: «A Theoretician's Introduction», *EATCS Bulletin*, 29 pp. 69–84, 1986.

[2] BROOKES, S., HOARE, C., and ROSCOE, A. — «A Theory of Communicating Sequential Processes», *Journal of the ACM*, 31:7, pp. 560-599, 1984.

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

[4] DAHL, O.-J., MYHRHAUG, B., and NYGAARD, K., *SIMULA 67, Common Base Language*, Norwegian Computer Center, 1967.

[5] DAYAL, U., and DITTRICH, K. (eds.) — *Proceedings of the International Workshop on Object-Oriented Database Systems*, Los Angeles, IEEE Computer Society (1986).

[6] EHRICH, H.-D., and SERNADAS, A. — «Algebraic Implementation of Objects over Objects», *Proceedings of the REX Workshop on Stepwise Refinement of Distributed Systems: Models, Formalisms, Correctness*, Springer Verlag (in print).

[7] EHRICH, H.-D., SERNADAS, A., and SERNADAS, C. — «Abstract Object Types for Databases», *Advances in Object-Oriented Database Systems*, K. Dittrich (ed), pp. 144–149, Springer Verlag, 1988.

[8] EHRICH, H.-D., SERNADAS, A., and SERNADAS, C. — «Objects, Object Types and Object Identity», *Categorical Methods in Computer Science with Aspects from Topology*, H. Ehrig *et al.* (eds.), LNCS 393, pp. 142–156, Springer Verlag, 1989.

[9] EHRICH, H.-D., SERNADAS, A., and SERNADAS, C. — «From Data Types to Object Types», *Journal of Information Processing and Cybernetics*, EIK 26(1/2), pp. 33–48, 1990.

[10] FIADEIRO, J., and SERNADAS, A. — «Logics of Modal Terms for Systems Specification». *Journal of Logics and Computation* (in print)..

[11] GOLDBERG, A., and ROBSON, D. — *Smalltalk 80: The Language and its Implementation*, Addison-Wesley, 1983.

[12] GOLDBLATT, R. — *Topoi, the Categorial Analysis of Logic*, North-Holland, 1979.

[13] HENNESSY, M. — *Algebraic Theory of Processes*, MIT Press, 1988.

[14] HENNESSY, M., and MILNER, R. — «Algebraic Laws for Nondeterminism and Concurrency», *Journal of the ACM*, 32:1, pp. 137–161, 1985.

[15] HEWITT, C. — «Viewing Control Structures as Patterns of Passing Messages», *Journal of Artificial Intelligence*, 8:3, pp. 323–364, 1977.

[16] HEWITT, C., and BAKERR, H. — «Laws for Communicating Parallel Processes», *Proceedings of the 1977 IFIP Congress*, pp. 987–992, IFIP, 1977.

[17] LOCHOVSKI, F. (ed.) — «Special Issue on Object-Oriented Systems», *IEEE Database Engineering* 8:4, 1985.

[18] SERNADAS, A., FIADEIRO, J., SERNADAS, C., and EHRICH, H.-D. — «Abstract Object Types: A Temporal Perspective», *Proceedings of the Colloquium on Temporal Logic and Specification*, B. Banieqbal, H. Barringer and A. Pnueli, (eds.), pp. 324–350, Springer Verlag, 1989.

[19] SERNADAS, A., SERNADAS, C., and EHRICH, H.-D. — «Object-Oriented Specification of Databases: An Algebraic Approach», *Proceedings of the 13th VLDB Conference*, P. Hammersley (ed.), pp. 107–116, VLDB, 1987.

[20] SHRIVER, B., and WEGNER, P. (eds.) — *Research Directions in Object Oriented Programming*, MIT Press, 1987.