A Categorial Framework

for

Process Interconnection

J.F.Costa and A.Sernadas

INESC
Apartado 10105, 1017 Lisboa Codex, PORTUGAL

Phone: 351-1-3523870
Telefax: 351-1-525843
E-mail: {fgc,acs}@inesc.pt

Abstract - *A categorial semantics for processes and communities of concurrent processes is presented. The model covers complex processes that arise by composition of more elementary processes and provides a deeper understanding of event sharing and event calling. A category of processes is defined that is shown to be complete (with limits reflecting parallel composition) and cocomplete (with colimits reflecting choice).*

Introduction. Several convenient, sophisticated mathematical models for processes have been proposed, such as those reported in [3,11,12,14], taking some or all of the basic three approaches: axiomatization, denotational semantics and operational semantics. Many relevant issues, like nondeterminism and limitations of the interleaving models, have been clarified and are by now well understood. As a whole, the basic notion of process is mathematically well founded. This nice state of affairs is reflected in Hennessy's and Olderog's *textbooks* [11,14].

The model of processes we adopt within this paper is obtained from the traditional *trace model* (PS model, from *prefix-closed set*) by dropping the prefix-closure condition. Let us consider a chocolate vending machine VM=μx.coin•!choc•x. The traditional semantics of such a process term is the pair $\langle E,\Lambda\rangle$ with E={coin,choc} and Λ is a subset of E^* denoted by the regular expression (coin choc)*(coin + ε). But, this semantics provides no information about the process liveness (indicated by the !), namely, that a chocolate must always be delivered by the machine once a coin is inserted.

The idea of quiescence, developed in [13] in the context of a logic to reason about specifications, and developed in [4,5] within an algebraic framework, provides a solution for the problem of representing state-dependent liveness in a process. We say that a process (or a community of processes) is in a *quiescent state* iff it is not able to produce any action by itself unless triggered by the outside. Returning to our example, the quiescent traces of the VM 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)*. If we are willing to include recursive definitions, we have to consider infinite traces, because an infinite trace is quiescent. For instance, a CLOCK=μx.!tick•x that produces a nonterminating tick signal cannot be in a quiescent state, but could be represented by an infinite trace tick$^{\omega}$. However only short term commitments will be considered in this paper (see [6] for completeness).

A process is represented by the set of its quiescent traces. And a trace of the community is quiescent iff it can be projected onto a quiescent trace of every of its components. Indeed, the whole community is in a quiescent state iff all its components are in a quiescent state.

According to the model just described, which we denote by QS (from *quiescent set*), given a finite set of events (the *alphabet*) E, any pair <E,Λ>, where Λ is a set of traces over E, is a *process*. For instance, the pair <{coin,choc},(coin choc)*> 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 liveness or passive, since it is always quiescent. Those QS-processes that are not PS-processes display some liveness. Indeed, we say that the process <{coin,choc},(coin choc)*> above is active in the sense that after coin, for instance, it is willing to make choc happen.

Starting from basic processes we build more complex processes, viewing the world in general and the computer systems in particular as communities of interacting, fully concurrent processes. It is significant that the relationships between processes can be well understood within a categorial framework. In previous work it was possible to explain aggregation of concurrent processes [7,8], symmetrical (event sharing) interaction [7,8,15] and non symmetrical (event calling) interaction [15].

The main ideas on categorial fundamentals of process theory could be found in [7,8,9,10,16]: (a) morphisms express inheritance, ie, a process Q inherits the behaviour of a process P iff there is a process morphism f:Q$\rightarrow$P; (b) communities of processe are seen as diagrams in a suitable category; (c) joint behaviour is limit.

Parallel composition with sharing of parts can be achieved by constraining concurrency. Categorially we fulfil such constraints by means of a pullback construction. In viewing the events of a distributed computation, we ascribe precise places and times to all the events of interest, attaching places to processes. When two events take place at the same time and place we say that their occurrence is synchronous. Thus, synchronous occurrences represent cooperative effects.

We shall consider only two forms of interaction: event sharing and event calling. These two basic forms of process interaction have been found to be sufficient in practice. 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 customers of the VM to be able to eat chocolates, delivered by the machine, without making a coin made by the first customer also a coin in the life of the second customer and vice-versa!

First we introduce a category of processes with liveness requirements and show that its limits reflect parallel composition. Special attention is dedicated to the parallel composition operation in order to explain the liveness of a composite process. It is also showed that colimits reflect choice. Then, we deal with communities of processes in the presence of event sharing and event calling. Since interleaving semantics should be retrieved from more general semantic domains, we explain how to use diagrams in order to be able to derive the properties of processes with liveness requirements in the context of the interleaving paradigm.

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* [1].

Process morphisms. The category Pfn of *partial functions* is defined as Set but with partial functions as morphisms. It has sets, as before, as objects, but more morphisms than Set. Therefore, Set is a wide subcategory of Pfn. The isomorphisms of Pfn are the (total) bijections. Furthermore, $\varnothing$ 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.

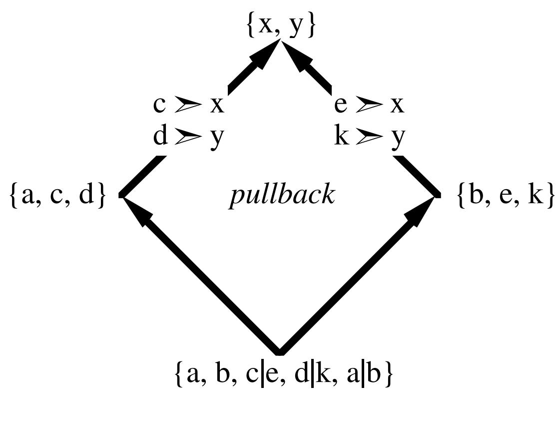

Fig. 1. A limit within Pfn

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 small cocomplete. Figs. 1 and 2 will help the reader to get acquainted with pullbacks and pushouts in Pfn.

$$
\begin{tikzcd}
 & \{x, y\} \arrow[dl, "x \succ c \\ y \succ d"'] \arrow[dr, "x \succ e \\ y \succ k"] & \\
\{a, c, d\} \arrow[dr] \arrow[rr, phantom, "\textit{pushout}"] & & \{b, e, k\} \arrow[dl] \\
 & \{a, x, y, b\} &
\end{tikzcd}
$$

Fig. 2. A colimit within Pfn

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

Given a process P=<E,Λ> we refer to E as alph(P) — the alphabet of P — and we refer to Λ as lang(P) — the quiescent language of P.

A partial map h:A→B can be canonically extended to h*:A*→B* as follows: (i) h*(ε)=ε and (ii) h*(as)=if h(a)↓ then h(a)h*(s) else h*(s). By h(a)↓ we mean that h is defined at a. Otherwise we say that h is undefined and we write h(a)↑. 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. A *process morphism* $h{:}\langle E_1,\Lambda_1\rangle \to \langle E_2,\Lambda_2\rangle$ is a partial event map $h_E{:}E_1 \to E_2$ such that $h_E^*(\Lambda_1) \subseteq \Lambda_2$.

A *concrete category* over a base category B is a pair <C,U>, where C is a category and U:C→B is a faithful functor.

Proposition. The processes and process morphisms constitute a concrete category Proc over Pfn.

Proof: Let $f{:}P_1 \to P_2$ and $g{:}P_2 \to P_3$ 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:Proc→Pfn that maps each process onto its event alphabet. □

Proposition. The category Proc has a terminal object.

Proof: Take $T=<\varnothing,\{\varepsilon\}>$. For every process $<E,\Lambda>$, the completely undefined map $\perp_E:E\rightarrow\varnothing$ is a process morphism $\perp:<E,\Lambda>\rightarrow T$ because $\perp_E^*(\Lambda)\subseteq\{\varepsilon\}$. □

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,

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

An elements $a|b$ of $\mathrm{alph}(P)\times_{Set}\mathrm{alph}(Q)$ 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,

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

A *source* in a category C is a pair $<X,\{f_i\}_{i\in I}>$, usually denoted by $(f_i:X\rightarrow X_i)_{i\in I}$, consisting of an object X and a family of morphisms $f_i:X\rightarrow X_i$ with domain X, indexed by some class I. A source $(f_i:X\rightarrow X_i)_{i\in I}$ in a concrete category <C,U> over B is called *initial* provided that a B-morphism $f:UY\rightarrow UX$ is a C-morphism whenever each composite $f_i\circ f:UY\rightarrow UX_i$ is a C-morphism. Let <C,U> be a concrete category over B. A limit L of a diagram $D:I\rightarrow C$ is called a *concrete limit* of D in <C,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 <C,U> is a concrete category over B and $D:I\rightarrow C$ is a diagram then $L=(f_i:L\rightarrow D_i)_{i\in |I|}$ is a concrete limit in <C,U> iff U(L) is a limit of $U\circ D$ and L is an initial source in <C,U>.

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.

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

Proof: With $\Lambda=\{s\in E^*$: for every $i\in I$, $f_{iE}{}^*(s)\in\Lambda_i\}$, 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 $f_i \circ g$ is a process morphism then $g:\langle E',\Lambda'\rangle\rightarrow\langle E,\Lambda\rangle$ is a process morphism. Since, for every $i\in I$, $f_{iE}{}^*(g_E{}^*(\Lambda'))\subseteq\Lambda_i$ 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 $\Lambda=\{s\in E^*$: for every $i\in I$, $f_{iE}{}^*(s)\subseteq\Lambda_i\}$, if we take E to be the resulting alphabet of the whole community.

Proposition. Proc is finitely complete.

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

Proposition. U lifts limits uniquely.

Proof: U lifts limits. It is straightforward to show that <Proc,U> 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 $Uf:UP\rightarrow E$ is a bijection then there can only exist one Λ such that $\langle E,\Lambda\rangle$ is a process and $f_E:\mathrm{alph}(P)\rightarrow E$ and $f_E{}^{-1}:E\rightarrow\mathrm{alph}(P)$ are process morphisms: lang(P).

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.

Proposition. The category Proc has an initial object.

Proof: Take $I=\langle\varnothing,\varnothing\rangle$. For every process $\langle E,\Lambda\rangle$, the empty map $\varnothing_E:\varnothing\rightarrow E$ is a process morphism $\varnothing:\langle\varnothing,\varnothing\rangle\rightarrow\langle E,\Lambda\rangle$ because $\varnothing_E{}^*(\varnothing))\subseteq\Lambda$. □

In order to apply the technique of lifting colimits of Pfn onto colimits of Proc, we need:

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

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

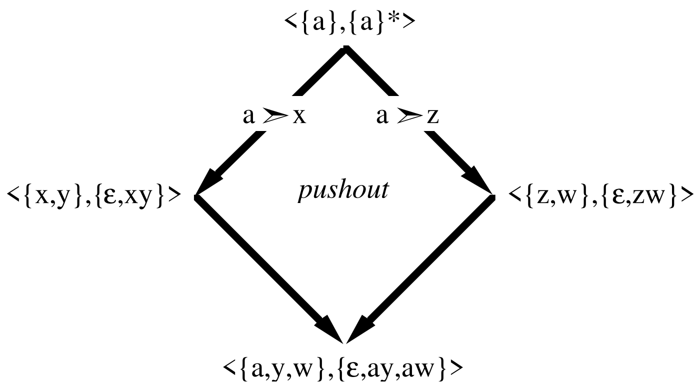

Fig. 3. Illustration of choice by pushing out.

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

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

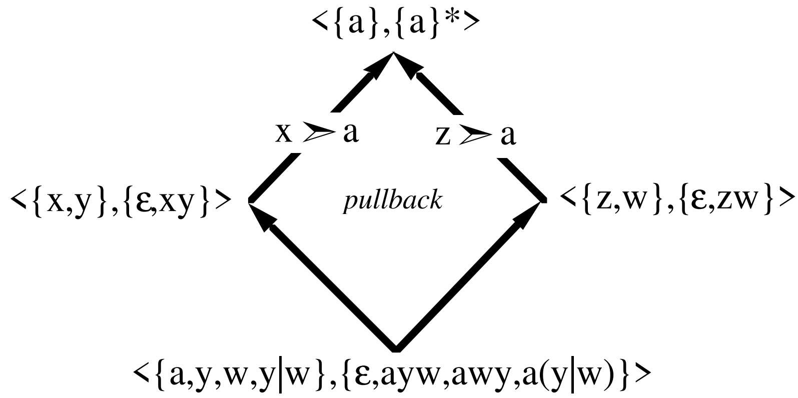

Fig. 4. A partial, cooperative parallel composition of two processes.

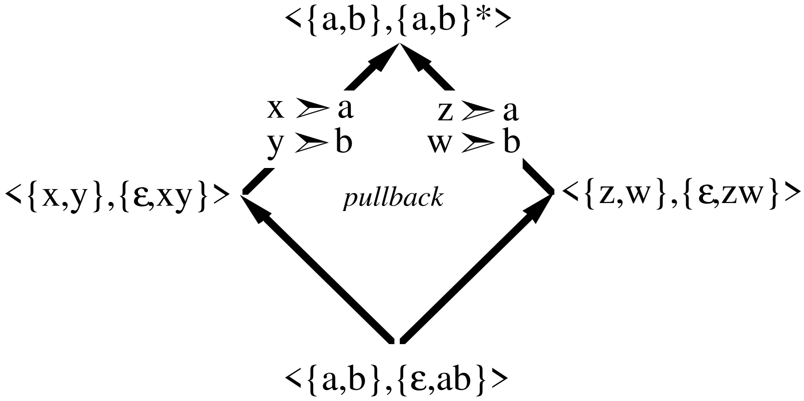

Fig. 5. A total, cooperative parallel composition of two processes.

Consider fig. 4. Let us calculate the pullback process P||$_R$Q, where R=<{a},{a}*>. 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*|*y*) 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*|*y* but not the other.

As processes have local alphabets, the interaction of two given processes, say P and Q, must be specified by a restriction map (symmetric wrt both α(P) and α(Q)) to the alphabet of P×Q. This map has the property of being injective. The process P||Q is then obtained by taking the product of P and Q and then providing an initial lift of the event map ϕ to Proc. In that circumstances, we can show that there exists a middle object R and morphisms $P \xrightarrow{f} R \xleftarrow{g} Q$ such that the initial lift of ϕ is an equalizer of the pair $R \underset{g}{\overset{f}{\leftleftarrows}} P\times Q$. Thus, the notion of process synchronisation has a categorial counterpart in the limit operation or in an equivalent lifting procedure. Limiting pullback diagrams or lifting products stand for process synchronisation in pairs.

But asymmetrical forms of interaction are equally important. That is the case if we want two different processes to be able to engage independently in an event of a third process. In this case the interaction is said to be by event calling. The situation can be alternatively handled by limiting another kind of *twin peaks* diagram or by lifting the product. Limiting a *twin peaks* diagram to give semantics to the call mechanism is a procedure very close to that used before for capturing the interleaving semantics of the parallel composition. Fig. 6 illustrates the construction. Thus, such *twin peaks* diagrams stand for process calling in pairs. This idea can be generalised to communities of processes that interact by event calling.

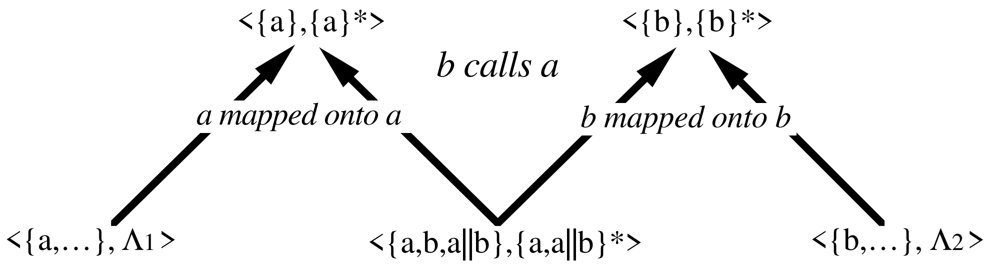

Fig. 6: Two processes interacting by event calling.

The composition of two processes P and Q in the presence of single event calling (say *b calls a*) is easily obtained. We denote the composite process by $P\|_{b>>a}Q$. Consider the following three processes: VM, 1.CUST and 2.CUST. The VM has been introduced before. The other two processes are isomorphic copies of CUST=!coin.choc.**stop**. The main interest of the *twin peaks* diagram over a call like $coin_1$>>coin by process 1.CUST and VM is to make sure that every $coin_1$ event (of 1.CUST) implies a coin event (of VM), but not vice-versa. Otherwise, we would also identify $coin_1$ with $coin_2$, that is, the two customers interacting via the VM would have their events synchronized! With event calling we avoid the symmetry of event sharing. In this case (considering only the coin events for the moment) what we want is the limit of the abbreviated diagram of fig. 7. What we want is to make sure that whenever $coin_1$ happens the effects of coin should take place (and similarly for $coin_2$). By effects we mean consequences on further event interactions.

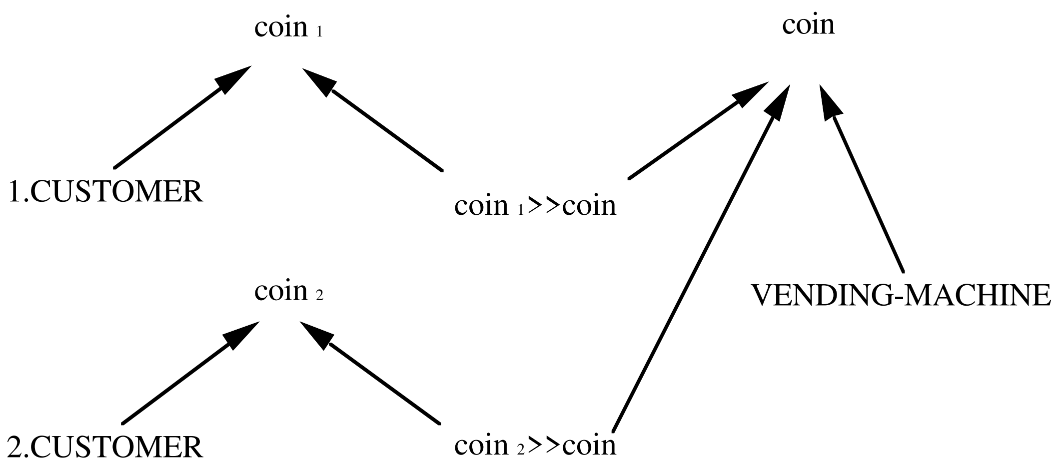

Fig. 7: Multiple event calling.

Of course, we can prove that mutual event calling by two processes amounts to event sharing.

Concluding Remarks. We presented a category of processes, Proc, displaying deterministic behaviour and liveness requirements. Proc is finitely complete and cocomplete: limits reflect parallel composition and colimits reflect choice. Within Proc we explained the different kinds of interaction as event sharing (standing for CSP-like symmetrical interaction) and event calling (an ACP-like asymmetrical interaction).

Acknowledgements. This work was partially supported by the Esprit Basic Research Action 3023 (IS-CORE) and by the JNICT Project PMCT/C/TIT/178/90 (FAC3).