Linearity of Client/Server Systems

G.Degli Antoni, D.Cabianca, M.Vaccari, M.Benini, F.Casablanca

Università degli Studi di Milano

April 24, 1995

Abstract

An important property of a client/server system is the independence of the interaction between the server and a client from the interactions that the server might have with other possible clients. In this paper we try to give a definition of such a property, that we called *linearity*, by means of the Process Algebra formalism. Through some examples of linear and non linear systems, we illustrate the expressiveness of our definition. Moreover, we present a sufficient condition for linearity and discuss the preservation of linearity w.r.t. process-algebraic operators applied to linear systems.

1 Introduction

The growth of computer networks and distributed computing has led to the development of the client/server paradigm, which is widespread nowadays. A client/server system is a composite system which allows distributed computing, analysis and presentation [Si92]. The client is a process (program) that sends a message to a server process (program) requesting that the latter performs a task (service). The server process (program) fulfills a client's demand by performing the requested task. Henceforth the need of a conceptual framework and tools which address the problem of the reliability of such systems is strongly felt. For example, a very important property of a client/server system is that the interaction of the server with a client is not harmful to another client, i.e. does not produce any alteration in the interaction of the latter client with the server. In analogy with physical systems, we call this property *linearity*.

We believe that Process Algebra is a suitable framework in which studying client/server systems and their properties, for its descriptive power, formal elegance and variety of established results. Moreover, it has been successfully applied to real-life systems for specifying communication protocols[Br88], in VLSI design [Mi85] and fault-tolerant systems [Pr87]. A regrettable feature is the proliferation of a variety of alternative systems, which differ only in minor formal details. Among them, we chose to use Hoare's CSP, because it is more focused toward the use of process-algebraic tools for defining and verifying specifications of concurrent and distributed systems. Properties of concurrent systems have usually been classified in liveness and safety properties. Linearity is a liveness property, which resembles, for example, deadlock-freeness, but it can not be reduced to it or to other known properties, because it depends inherently on the structure of the system.

Process Algebra specialists will recognize an analogy with the property of compositionality. Compositionality states that the semantics of a process is context-independent and can be obtained by means of a function on the semantics of its subprocesses. There is a clear, albeit indirect, connection between compositionality and linearity. However, linearity is a weaker property which concentrates on the services requested by the client and/or supplied by the server and hence on the structure of the system.

In this paper we introduce a formal definition of client/server systems (section 2) and of the property of linearity (section 3). Through some examples, we discuss the expressiveness of this concept (section 4). A sufficient condition for linearity is introduced (section 5). Finally we prove that some process-algebraic operators preserve linearity, i.e. when applied to linear systems they produce new linear systems (section 6).

2 Process-Algebraic View

In this section we will give a short introduction to CSP focused on definitions and concepts useful for our work. More exhaustive information can be found in [Ho85]. The definition of client/server systems will follow.

2.1 A Process Algebra : CSP

Process algebras represent processes by means of a set of actions (events), operators and process variables. The operators act as process combinators to build new processes from existing ones.

Some alternative models have been proposed, which differ in the operators and in the semantic description adopted. The main CSP operators and syntactic requirements are specified formally in the following definition.

Definition 2.1 *Let* $\mathbf{Act} = \{a, b, \ldots\}$ *an infinite set of actions, and* $f\colon \mathbf{Act} \to \mathbf{Act}$ *a relabeling function. Let* $\mathbf{AVar} = \{x, y, \ldots\}$ *a countable set of action variables. Let* $\mathbf{PVar} = \{X, Y, \ldots\}$ *a countable set of process variables. Let* $A \subseteq \mathbf{Act}$*. The set of process expressions* **Proc** *is the smallest set which include all terms of the following grammar:*

$$\begin{aligned} P ::= \;& a \to P \mid (a \to P \,|\, b \to Q) \mid x\colon A \to P(x) \mid P \,|||\, Q \mid P \,\|\, Q \mid P \,\Box\, Q \mid \\ & P \sqcap Q \mid P \backslash A \mid f(P) \mid X \mid \mu X\colon A.F(X) \end{aligned}$$

In the final term the variable X is bound. The free variables $\mathsf{fv}(P)$ of a process P are those which occur unbounded in an expression. If $\mathsf{fv}(P) = \emptyset$ we say that P is *closed*. The set of closed CSP processes is denoted by **Cproc**.

The behavior of a process P depends also on its *alphabet* (denoted αP), which is the set of possible actions for the process, not necessarily coinciding with the set of actions appearing in the expression denoting the process. For this reason a *process* is fully specified only when, besides its expression, also its alphabet is given. Here is the intended meaning of the CSP operators; for a more formal treatment, based on an axiomatization of the operators, we refer to [Ho85]. Such axioms are very useful to establish equivalences and to reason about processes.

- $a \rightarrow P$(*prefix*): a process which offers action a and after behaves as the process P;
- $a \rightarrow P \mid b \rightarrow Q$(*deterministic choice*; generalized to $(x{:}\, A \rightarrow P(x))$): a process which offers a and then commits to P, or offers b and then commits to Q;
- $P \;|||\; Q$(*interleaving*): a process which behaves as the interleaved shuffle of processes P and Q;
- $P \sqcap Q$(*nondeterministic choice*): a process which behaves as P or Q, independently by the consequences on the environment;
- $P \,\Box\, Q$(*general choice*): a process which behaves as a deterministic choice if different actions are offered by its operands and as a nondeterministic choice if identical actions are offered;
- $P \backslash A$(*concealment*): a process which behaves like P, but treats the actions in A as local "hidden" actions;
- $P \,||\, Q$(*concurrency*): a process which behaves as the interleave operator for actions not shared by their alphabets and as a synchronization (communication) for shared actions (P and Q have to offer the same shared action to proceed, and deadlock otherwise).
- $f(P)$(*change of symbol*): a process which behaves like P, but which outputs action $f(a)$ instead of a, for any $a \in \alpha P$.
- $\mu X{:}\, A.F(X)$(*recursion*): a process which behave like $F(X)$, substituting itself to the process variable X when needed.

Some processes, which have a peculiar behavior, have been given names:

- RUN_A is the process which can engage in any event of its alphabet A at all times:
$$\alpha\mathsf{RUN}_A = A; \quad \mathsf{RUN}_A = x{:}\, A \rightarrow \mathsf{RUN}_A;$$
- STOP_A is the process which does not engage in any action:
$$\alpha\mathsf{STOP}_A = A; \quad \mathsf{STOP}_A = x{:}\, \emptyset \rightarrow P(x);$$

- CHAOS_A is the process which may behave as any process:

$$\alpha\mathsf{CHAOS}_A = A; \quad \mathsf{CHAOS}_A = \mu X\colon A.X.$$

Hoare was led to study concurrent systems by the wish to extend the formal methods he had devised for specifying and studying sequential computer programs.

In computer programming, by specification is usually understood a logical formula that expresses the initial and final conditions on the variables of the program [Fr92]. For concurrent systems it is not possible to apply directly this definition. Here the specification of a system (its relevant and observable aspects) are the sequences of events (the *traces*) that might happen [Ho85]. Hence a specification is a predicate $\Pi(tr)$ defined on traces. Dually, a process' semantics might be understood also as the set of its possible traces. The satisfaction of a specification by a process can be defined as follows.

Definition 2.2 *Given a process P and a predicate $\Pi(tr)$,*

$$P \ \mathsf{sat}\ \Pi(tr) \overset{def}{=} (\forall tr\colon tr \in \mathsf{Traces}(P)\colon \Pi(tr))$$

When the process is not deterministic, traces alone are not enough to specify desired behaviors. Sometimes it is useful to know if, after producing a trace, the process will stop or go on producing other actions. The set of actions which can not be produced after a trace is called its *refusal set.* Sets of trace-refusals pairs (*failures*), are an alternative description of a system, which leads to the following definition of satisfaction.

Definition 2.3 *Given a process P and a property $\Pi(fa)$,*

$$P \ \mathsf{sat}\ \Pi(fa) \overset{def}{=} (\forall fa\colon fa \in \mathsf{failures}(P)\colon \Pi(fa))$$

In the following we will need to consider traces or failures *restricted* to a given set of actions (alphabet). Here is the notation we adopt.

Definition 2.4 *Given $P \in \mathsf{Proc}$, $tr \in \mathsf{Traces}(P)$, $fa \in \mathsf{failures}(P)$, $A \subseteq \mathbf{Act}$,*

$$tr \upharpoonright A = \begin{cases} \epsilon & \textit{if } tr = \epsilon \\ a \cdot (x \upharpoonright A) & \textit{if } tr = a \cdot x \textit{ and } a \in A \\ (x \upharpoonright A) & \textit{if } tr = a \cdot x \textit{ and } a \notin A \end{cases}$$

$$fa \upharpoonright A = \langle t \upharpoonright A, ref \cap A\rangle \qquad \textit{where } fa = \langle t, ref\rangle$$

2.2 Server/Client Systems

In the following our universe will be the set **Cproc** of the closed CSP terms. A *client/server system* is a process such that:

- it is composed by a finite number of parallel processes;
- these processes can be conceptually split in a *server* process and in a set of *client* processes, mutually independent.

Definition 2.5 *A (client/server) system is a pair* $\underline{\mathcal{Z}} = \langle S, \{C_1, \ldots, C_n\}\rangle$*, where* S *is the server process, and every* C_i *is a client process. The following conditions must hold:*

- $(\forall i, j\colon 1 \leq i, j \leq n \wedge i \neq j\colon \alpha C_i \cap \alpha C_j = \emptyset)$
- $(\forall i\colon 1 \leq i \leq n\colon \alpha C_i \subseteq \alpha S)$

We will refer to the set of all systems as **Systems**.

We can define a small set of useful functions to make notation more concise:

Definition 2.6 *Given a system* $\underline{\mathcal{Z}} = \langle S, \{C_1, \ldots, C_n\}\rangle$*, the following functions are defined:*

$$\begin{array}{lcl}
\mathsf{Server}(\underline{\mathcal{Z}}) & \overset{def}{=} & S \\
\mathsf{Clients}(\underline{\mathcal{Z}}) & \overset{def}{=} & \{C_1, \ldots, C_n\} \\
\mathsf{Parts}(\underline{\mathcal{Z}}) & \overset{def}{=} & \{\mathsf{Server}(\underline{\mathcal{Z}})\} \cup \mathsf{Clients}(\underline{\mathcal{Z}}) \\
\alpha\underline{\mathcal{Z}} & \overset{def}{=} & (\bigcup P\colon P \in \mathsf{Parts}(\underline{\mathcal{Z}})\colon \alpha P)
\end{array}$$

A process underlies every system; it is simply the parallel composition of all parts of a system.

Definition 2.7 *Given a system* $\underline{\mathcal{Z}} = \langle S, \{C_1, \ldots, C_n\}\rangle$*, the associated process is:*

$$\mathsf{Proc}(\underline{\mathcal{Z}}) \overset{def}{=} (\| P\colon P \in \mathsf{Parts}(\underline{\mathcal{Z}})\colon P)$$

We note that the same process can be generated by more then one system. In the following, to make notation clearer, we may write just $\underline{\mathcal{Z}}$ in place of $\mathsf{Proc}(\underline{\mathcal{Z}})$.

2.2.1 Ordering of Systems

Systems can be ordered by means of a convenient relation.

Definition 2.8 *Given* $\underline{\mathcal{Z}}, \underline{\mathcal{S}} \in$ **Systems**,

$$\underline{\mathcal{Z}} \preceq \underline{\mathcal{S}} \textit{ iff } \mathsf{Server}(\underline{\mathcal{Z}}) = \mathsf{Server}(\underline{\mathcal{S}}) \wedge \mathsf{Clients}(\underline{\mathcal{Z}}) \subseteq \mathsf{Clients}(\underline{\mathcal{S}})$$

It is easy to see that $\preceq$ is a partial ordering.

Lemma 1 *Given a system* $\underline{\mathcal{Z}}$*:*

$$\{\underline{\mathcal{S}} \mid \underline{\mathcal{S}} \preceq \underline{\mathcal{Z}}\} = \{\underline{\mathcal{S}} \mid (\exists C\colon C \subseteq \mathsf{Clients}(\underline{\mathcal{Z}})\colon \underline{\mathcal{S}} = \langle \mathsf{Server}(\underline{\mathcal{Z}}), C\rangle)\}$$

and the minimum of such a set is the process $\langle \mathsf{Server}(\underline{\mathcal{Z}}), \emptyset\rangle$.

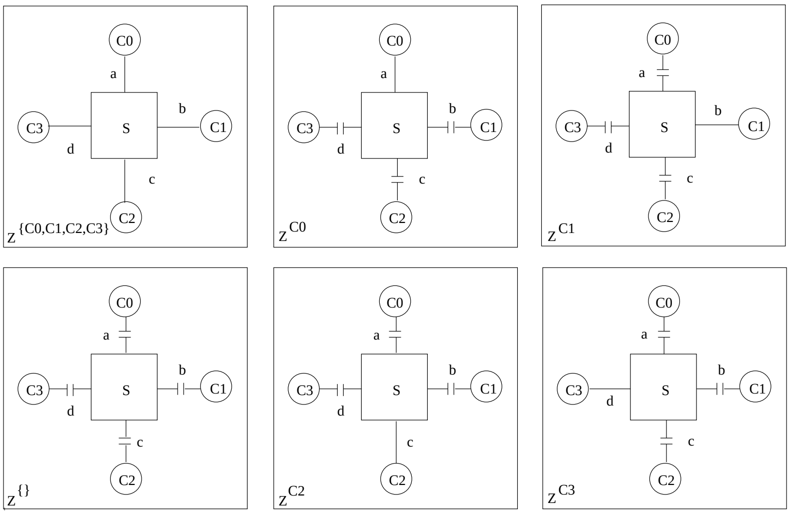

Figure 1: Examples of connection diagrams with cuts

2.3 A Graph View

Seeing a system as a graph is often more expressive than seeing it as a mathematical formula. In [Ho85] each system is represented by a *connection-diagram.* Given a client/server system, a definition of a corresponding graph is straightforward. For the sake of simplicity, the information on the client/server structure is kept.

Definition 2.9 (connection-diagram) *Given a system* $\underline{\mathcal{Z}}$*,*

$$\mathsf{graph}_{\mathsf{S/C}}(\underline{\mathcal{Z}}) \overset{def}{=} \langle \mathsf{Parts}(\underline{\mathcal{Z}}), W\rangle$$

s.t. $W = \{\langle \mathsf{Server}(\underline{\mathcal{Z}}), C, l\rangle \mid C \in \mathsf{Clients}(\underline{\mathcal{Z}}) \wedge l \in (\alpha C \cap \alpha(\mathsf{Server}(\underline{\mathcal{Z}})))\}$*. The set of all these graphs will be denoted with* $\mathbf{Graphs}_{\mathbf{S/C}}$*.*

According to the ordering of systems, and in particular with lemma 1, we can define the graphical operation of "cutting" an edge, as the generation of a subsystem.

Definition 2.10 *Let* $\underline{\mathcal{G}} = \langle\{S\} \cup Cl, \phi\rangle \in \mathbf{Graphs}_{\mathbf{S/C}}$*, the function:* $\mathsf{cut}_{\underline{\mathcal{G}}}$ *is defined as follows:*

$$\mathsf{cut}_{\underline{\mathcal{G}}}(Gl) \overset{def}{=} \begin{cases} \langle\{S\} \cup Gl, \psi\rangle & \langle S, Gl\rangle \preceq \langle S, Cl\rangle \wedge \psi = \{\langle S, C, l\rangle \in \phi \mid C \in Gl\} \\ \textit{undefined} & \textit{otherwise.} \end{cases}$$

From now on, we will make a liberal use of graphs without bothering to work out all the formal details.

Example 2.1 *Some subsystems of* $\underline{\mathcal{Z}} = \langle S, \{C_0, C_1, C_2, C_3\}\rangle$ *are represented by means of connection-diagrams in Fig. 1.*

3 Linear Server/Client Systems

The typical client/server interaction is started by the client, which asks the server for a service. In this sense, the server is subordinated to the client.

It is important that, from the point of view of a client, the server's interactions with other clients be completely opaque; the server's behavior is reliable if the client is sure to obtain, sooner or later, the desired answer. If the server of a system is reliable for all its clients, we say that the system is *linear.*

The interaction between server and client is expressed by a sequence of events (a trace) that belongs either to the server's behavior, either to the client's; all the possible interactions are expressed by a set of shared traces.

A process is finite if no recursion operator appears in its definition, and a finite system is one denoting a finite process. To specify the concept of "service provided" we need only maximal traces, if the process is finite. From an algebraic point of view, this is obvious because all the traces of a finite process have finite length, and the set of traces of a process is closed under the prefix operator. If the process is not finite, the concept of maximal traces is not well defined, so we have to look at "stepwise" behavior of the system.

Definition 3.1 ($\mathsf{Done}_{\mathsf{fin}}$) *Given* $\underline{\mathcal{Z}} \in$ **Systems** *finite*

$$\mathsf{Done}_{\mathsf{fin}}(\underline{\mathcal{Z}}) \overset{def}{=} (ref = \alpha\underline{\mathcal{Z}} \Rightarrow (\forall c : c \in \mathsf{Clients}(\underline{\mathcal{Z}}) : tr \upharpoonright \alpha c \in \mathsf{MaxTraces}(c)))$$

We notice that a maximal trace of a client corresponds to a trace of a CSP failure such that the alphabet of that client is contained in the refusal set.

Definition 3.2 ($\mathsf{Done}_{\mathsf{inf}}$) *Given* $\underline{\mathcal{Z}} \in$ **Systems**

$$\mathsf{Done}_{\mathsf{inf}}(\underline{\mathcal{Z}}) \overset{def}{=} (\forall c : c \in \mathsf{Clients}(\underline{\mathcal{Z}}) : \langle tr, ref \rangle \upharpoonright \alpha c \in \mathsf{failures}(c))$$

The free variables tr and ref will be bounded by the definition of satisfiability (Defs. 2.2,2.3). The following lemma guarantees the consistency of our definitions.

Lemma 2 *Given* $\underline{\mathcal{Z}} \in$ **Systems** *finite,*$\mathsf{Done}_{\mathsf{inf}}(\underline{\mathcal{Z}}) \Leftrightarrow \mathsf{Done}_{\mathsf{fin}}(\underline{\mathcal{Z}})$.

$\mathsf{Done}_{\mathsf{fin}}$ and $\mathsf{Done}_{\mathsf{inf}}$ predicates fully agree with the definition of CSP specification given in Sect.2. Now it is time to take full advantage of our definition of client/server system to give a specification which exploits the possibility of decomposing the system.

Definition 3.3 ($\mathsf{Linearity}$) *Given* $\underline{\mathcal{Z}} \in$ **Systems***,*

$$\mathsf{Linear}(\underline{\mathcal{Z}}) \overset{def}{=} (\forall c : c \in \mathsf{Clients}(\underline{\mathcal{Z}}) : \underline{\mathcal{Z}}^c \ \mathsf{sat}\ \mathsf{Done}(\underline{\mathcal{Z}}^c)) \Rightarrow (\underline{\mathcal{Z}})\ \mathsf{sat}\ \mathsf{Done}(\underline{\mathcal{Z}})$$

where $(\underline{\mathcal{Z}})^c = \langle \mathsf{Server}(\underline{\mathcal{Z}}), \{c\} \rangle$ *and* $\mathsf{Done}(\underline{\mathcal{Z}})$ *can be finite or not.*

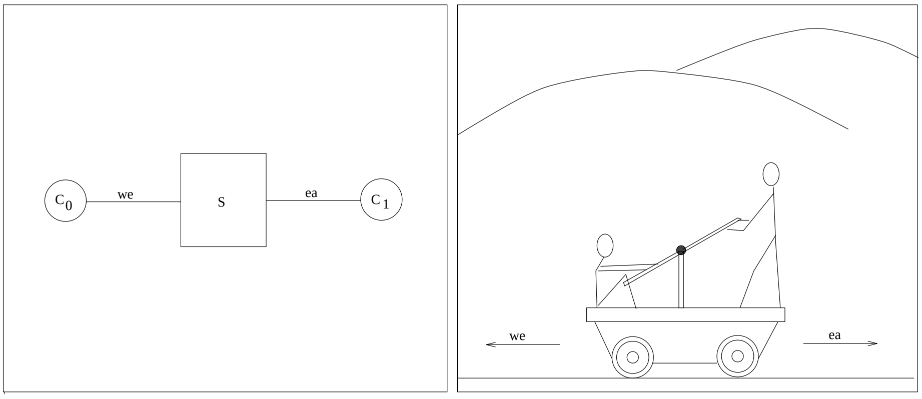

Figure 2: Connection-diagram of the wheel-cart system.

4 Some Examples

In this section we will study some systems by means of the concept of linearity. These example will also test the expressiveness of our definition.

4.1 The Wheel-cart

Our first example is a system given by two passengers on the sides of a wheel-cart running on a railway, each of them trying to move toward his direction acting on a lever. This is a client/server system, where the server is given by the wheel-cart and the clients are the two passengers. The system will be: $\underline{\mathcal{P}} = \langle S, \{C_0, C_1\}\rangle$, where S is the wheel-cart, whose possible actions are moving eastward or westward:

$$\alpha S = \{ea, we\} \qquad S = (we \rightarrow S) \sqcap (ea \rightarrow S)$$

The passengers try to force movement in single direction:

$$\begin{array}{ll} \alpha C_0 = \{we\} & C_0 = we \rightarrow C_0 \\ \alpha C_1 = \{ea\} & C_1 = ea \rightarrow C_1 \end{array}$$

It is very easy to prove that this system is linear using the following lemma.

Lemma 3 *Given a system* $\underline{\mathcal{S}} = \langle S, \{C_1, ..., C_n\}\rangle$,

$$(\forall i :: S \parallel C_i = \mathsf{Proc}(\underline{\mathcal{S}})) \Rightarrow \mathsf{Linear}(\underline{\mathcal{S}})$$

We could calculate the values of $S \parallel C_0$, $S \parallel C_1$ and $S \parallel C_0 \parallel C_1$. Proofs are very similar so, let's see only the first one:

Proof of $S \parallel C_0 = S$:

$$\begin{array}{cl} & S \parallel C_0 \\ = & \{\textit{definition of } S \textit{ and } C_0\} \\ & ((we \rightarrow S) \sqcap (ea \rightarrow S)) \parallel (we \rightarrow C_0) \\ = & \{\parallel \textit{ distributes over } \sqcap\} \\ & ((we \rightarrow S) \parallel (we \rightarrow C_0)) \sqcap ((ea \rightarrow S) \parallel (we \rightarrow C_0)) \\ = & \{\textit{laws on } \parallel\} \\ & (we \rightarrow (S \parallel C_0)) \sqcap (ea \rightarrow (S \parallel C_0)) \\ = & \{\textit{definition of } S\} \\ & S \end{array}$$

The value of all these processes is S, so, by lemma 3, our system is linear.

4.1.1 The finite wheel-cart

Let $\underline{\mathcal{P}}_{n,m} = \langle S, \{C_0^n, C_1^m\}\rangle$ be a variant of the previous system. The two systems have the same server, but the clients, which were infinite in $\underline{\mathcal{P}}$, are finite in $\underline{\mathcal{P}}_{n,m}$:

$$\alpha C_0^n = \{we\} \quad C_0^n = \underbrace{we \rightarrow \ldots \rightarrow we}_{\text{n times}} \rightarrow \mathsf{STOP}_{\{we\}}$$
$$\alpha C_1^m = \{ea\} \quad C_1^m = \underbrace{ea \rightarrow \ldots \rightarrow ea}_{\text{m times}} \rightarrow \mathsf{STOP}_{\{ea\}}$$

Here the clients are finite, so we can use Def. 3.1. Obviously, this system will behave exactly as $\underline{\mathcal{P}}$, as long the two clients can engage in an action. But, let's suppose C_0^n finish all its actions; its task is satisfied, according to our definition of $\mathsf{Done_{fin}}$. Now the server has to perform only ea actions, in order to perform C_1^m's task and to avoid deadlock. But this is impossible since the meaning of $\sqcap$, and, as a consequence, $\neg(\underline{\mathcal{P}}_{n,m}$ sat $\mathsf{Done}(\underline{\mathcal{P}}_{n,m}))$ and $\neg\mathsf{Linear}(\underline{\mathcal{P}}_{n,m})$.

4.1.2 The finite and deterministic wheel-cart

Let's develop another variant of the wheel-cart system: $\underline{\mathcal{Q}}_{n,m} = \langle S_{det}, \{C_0^n, C_1^m\}\rangle$, where

$$\alpha S_{det} = \{ea, we\} \qquad S_{det} = (we \rightarrow S_{det}) \,\square (ea \rightarrow S_{det})$$

By definition of deterministic choice, the actions of the server are determined according to the actions offered by clients. Then, when one of the two clients has completed his requests, only the actions of the other client are offered, the server is forced to produce these events, and this argument implies $\mathsf{Linear}(\underline{\mathcal{Q}}_{n,m})$.

The comparison between systems $\underline{\mathcal{P}}_{n,m}$ and $\underline{\mathcal{Q}}_{n,m}$ show clearly the relation intercurring between linearity and choice: with a nondeterministic server, we cannot guarantee linearity, while with a deterministic server, we get it.

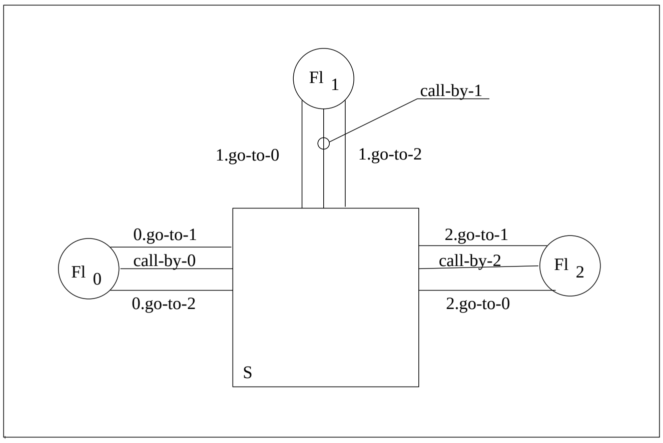

Figure 3: Connection-diagram of the elevator system, for $n = 3$.

4.2 The Elevator

Here we will show that the “footman” technique, devised by C. S. Scholten to solve the “dining philosophers” problem [Di65], can be used to make a linear system from a non-linear one.

Let’s take a system where the server is an elevator, while queues of users on each floor are the clients. Let’s suppose there are n floors.

We begin presenting the model of an user: he/she wants to go to floor k, and is waiting the elevator on floor i. Such an user can be represented by the process C_i^k:

$$\begin{aligned}\alpha C_i^k &= \{\mathit{call\_by\_i},\ \mathit{i.go\_to\_k}\}\\ C_i^k &= \mathit{call\_by\_i} \rightarrow \mathit{i.go\_to\_k} \rightarrow \mathit{Fl}_i\end{aligned}$$

and the queue at floor i is represented by Fl_i:

$$\begin{aligned}\alpha \mathit{Fl}_i &= \{\mathit{call\_by\_i}\} \cup \{\mathit{i.go\_to\_k} \mid 0 \leq k < n \wedge k \neq i\}\\ \mathit{Fl}_i &= \left(\Box k\colon 0 \leq k < n \wedge k \neq i\colon C_i^k\right)\end{aligned}$$

The action *call_by_i* means a call for the elevator from floor i, and the event *i.go_to_k* means the travel of an user from floor i to k.

The elevator doesn’t remember nor it schedules calls, so it can be described by:

$$\begin{aligned}\alpha S &= (\textstyle\bigcup i\colon 0 \leq i < n\colon \alpha \mathit{Fl}_i)\\ S &= (\Box i\colon 0 \leq i < n\colon \mathit{call\_by\_i} \rightarrow (\Box j\colon 0 \leq j < n \wedge i \neq j\colon \mathit{i.go\_to\_j} \rightarrow S))\end{aligned}$$

Its meaning is simple: when it receives a call from floor i, it goes to that floor, and then it moves itself to floor j, and there it stays, waiting for another call.

Putting all together, our system will be $\underline{\mathcal{L}}_n = \langle S, \{\mathit{Fl}_i \mid 0 \leq i < n\}\rangle$.

It is obviously true that $(\forall c\colon c \in \mathsf{Clients}(\underline{\mathcal{L}}_n)\colon \underline{\mathcal{L}}_n^c\ \mathsf{sat}\ \mathsf{Done}(\underline{\mathcal{L}}_n^c))$, but, in general (i.e. for any n), $\underline{\mathcal{L}}_n\ \mathsf{sat}\ \mathsf{Done}(\underline{\mathcal{L}}_n)$ does not hold.

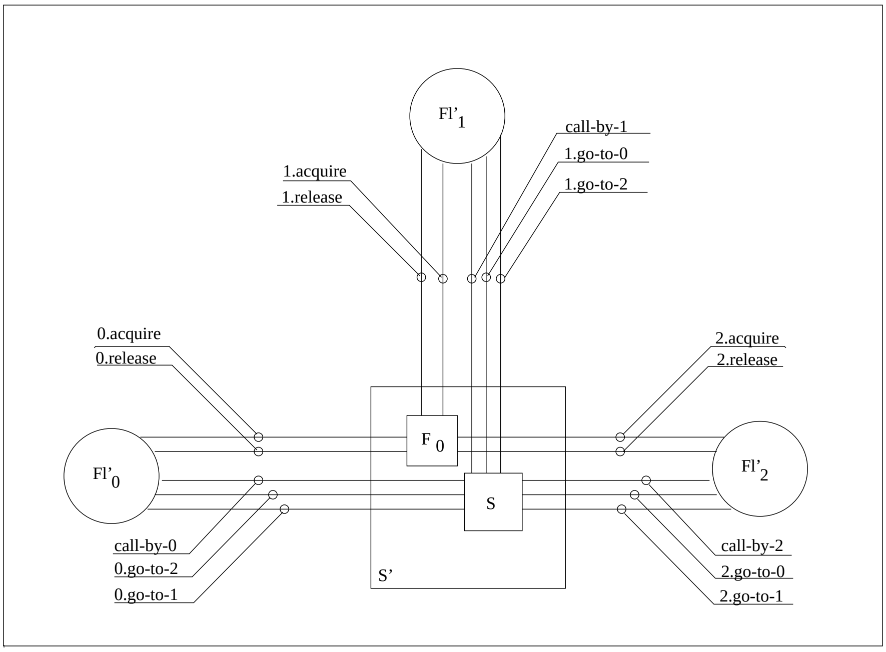

Figure 4: Connection-diagram of the linearized elevator system, for $n = 3$

Let's suppose that two users call the elevator from two different floors at the same time. A possible result will be:

$$\mathit{call\_by\_0} \rightarrow \mathit{call\_by\_1} \rightarrow ((\Box \ldots) \parallel (\Box \ldots))$$

the elevator will process the first event, but it will deadlock on the second, because it expects a *i.go_to_k* action, so we deduce that the system cannot be linear.

The "footman" technique suggests a solution to overcome non-linearity.

We have a problem with the clients because the elevator can serve only one user at a time, and we have a problem during the service of an user because it cannot accept any request from other users. To enforce such behavior we will extend the server with a "button" which commits the elevator to a single user; at the same time we will forbid to all clients but one to access the elevator (only one user can "push the button"); on completion of the service the elevator will be released and a new client may use the elevator.

For this purpose, alphabets of our processes have to be extended and their behavior needs to be slightly variated.

The "button" is defined as follows:

$$\begin{array}{ll} \alpha F_0 = D \cup E & F_0 = (x : D \rightarrow F_1) \\ \alpha F_1 = D \cup E & F_1 = (x : E \rightarrow F_0) \end{array}$$

where: $D = (\bigcup : 0 \leq i < n : \{i.acquire\})$ and $E = (\bigcup : 0 \leq i < n : \{i.release\})$.

The new system will be:

$$\underline{\mathcal{L}}' = \langle S', \{Fl'_i \mid 0 \leq i < n\}\rangle$$

where:

$$\alpha S' = \alpha S \cup \alpha F_0 \qquad S' = S \parallel F_0$$
$$\alpha Fl'_i = \alpha Fl_i \cup \{i.acquire, i.release\}$$
$$Fl'_i = i.acquire \rightarrow (\Box j\colon 0 \leq j < n\colon call\_by\_i \rightarrow i.go\_to\_j \rightarrow i.release \rightarrow Fl'_i)$$

Now it is straightforward to notice that no client interferes with another client's service; if one or more queues are available, the elevator commits nondeterministically to one of them. It is simple to prove that $\underline{\mathcal{L}}'$ **sat** $\mathsf{Done}(\underline{\mathcal{L}}')$ and so is obvious that $\mathsf{Linear}(\underline{\mathcal{L}}')$.

5 A Sufficient Condition for Linearity

In this paragraph we introduce an alternative definition of linearity which is given directly on the failure semantics of the process. It is possible to prove that this definition implies the former one and hence can be seen as a sufficient condition for linearity on the semantics for finite systems.

The concept of linearity is based on the intuition that the interaction between the server and a client should be independent from the presence of other clients. From a semantical point of view, this means that the semantics of any pair server/client should be equivalent to the semantics of the whole system *restricted* to the alphabet of the client.

Definition 5.1 *Given* $\underline{\mathcal{P}} \in$ **Systems**,

$$\mathsf{Linear}_{\mathsf{Fail}}(\underline{\mathcal{P}}) \overset{def}{=} (\forall c\colon c \in \mathsf{Clients}(\underline{\mathcal{P}})\colon \mathsf{failures}(\underline{\mathcal{P}}^c) \upharpoonright \alpha c = \mathsf{failures}(\underline{\mathcal{P}}) \upharpoonright \alpha c)$$

Theorem 5.1 *If* $\underline{\mathcal{Z}} \in$ **Systems** *is finite, then* $\mathsf{Linear}_{\mathsf{Fail}}(\underline{\mathcal{Z}}) \Rightarrow \mathsf{Linear}(\underline{\mathcal{Z}})$.

Here the property is expressed for the failure semantics, but the discussion has shown that it is actually parametric to the semantics.

6 Process-algebraic Operators and Linearity

When a property is defined in the context of Process Algebra, it is useful to determine if the property is preserved by a process-algebraic operator. If the answer is positive, the system built applying the operator to subsystems which have the property, will also join the property. Here the main CSP operators will be examined and the behavior w.r.t. linearity will be discussed.

- *Prefix.* If L is a linear system, for any action a, $a \rightarrow L$ will be linear if and only if the system has only one client, because prefix distributes over the parallel operator. However, if the system has more than one client, the principle of independence of clients is violated and $a \rightarrow L$ is not even a client/server system.

  Lemma 4 *If* $\mathsf{Linear}(L)$, *for* $a \in \mathbf{Act}$, $a \rightarrow L$ *is linear iff* $\mathsf{Clients}(L) = \{C\}$.

- *Nondeterministic/deterministic choice.* If L_1 and L_2 are linear, $L_1 \sqcap L_2$ and $L_1 \square L_2$ are not linear because it is not always possible to decompose the system in server and clients.

- *Parallelism.* If L_1 and L_2 are linear systems, $L_1 \,||\, L_2$ is a client/server system, because, for the laws of $||$, we will have:

$$S_1^1 \,||\, C_1^1 \,||\, \ldots \,||\, C_{N_1}^1 \,||\, S_1^2 \,||\, C_1^2 \,||\, \ldots \,||\, C_{N_2}^2 = S_1^1 \,||\, S_1^2 \,||\, C_1^1 \,||\, \ldots \,||\, C_{N_2}^2$$

  for which:

  - $\mathsf{Server}(L_1 \,||\, L_2) = S_1^1 \,||\, S_1^2$
  - $\mathsf{Clients}(L_1 \,||\, L_2) = \{C_1^1, \ldots, C_{N_1}^1, C_1^2, \ldots C_{N_2}^2\}$.

  However, not all the systems so obtained are linear and we have to state a supplementary condition for linearity. Here is the simplest one.

  Lemma 5 *If* L_1 *and* L_2 *are linear,* $L_1 \,||\, L_2$ *is linear if* $\alpha(L_1) \cap \alpha(L_2) = \emptyset$.

- *Relabeling.* If L is a linear system and f is a injective relabeling function, then $f(L)$ is a client/server system because:

$$f(S \,||\, C_1 \,||\, \ldots \,||\, C_N) = f(S) \,||\, f(C_1) \,||\, \ldots \,||\, f(C_N)$$

  Lemma 6 *If* L *is a linear system and* f *is an injective relabeling function,* $f(L)$ *is linear.*

- *Concealment.* If L is a linear system and *FA* a set of forbidden actions, the system $L \setminus FA$ is always linear, unless some client degenerates to the CHAOS process; in that case all the system will degenerate to CHAOS.

7 Conclusion

We would like to close with a brief discussion on the issue of "structured specifications". Our treatment of linearity has shown the need of a specification pattern where both the structure and the behavior of a system are described. Process-algebra specialists might be sceptical on the use of a kind of information (the structural one) which can not be "observed" through the system's behavior, but which has to be known a priori. Our reply is twofold. We are interested in real-life systems; especially in the case of system design, it is not too farfetched to suppose that the structure of the system is known. Moreover, recent approaches in process-algebra try to capture "observationally" not only the actions of a process, but also its structure ([FM91, BCH92]). From our point of view these proposals are at the moment too much concerned with the algebraic aspects of the theory and are not enough expressive. In a future we believe however that these theories will be the natural framework in which works as ours could be inserted and developed.

Acknowledgement. This work is a result of the cooperation between Università degli Studi di Milano, SGS-Thomson, and Consorzio Milano Ricerche.

References

[BK85] J. A. Bergstra and J. W. Klop. *Process Algebra for communication with Abstraction* Journal of Theoretical Computer Science, vol.77, Elsevier, 1985

[BCH92] G. Boudol, I. Castellani, M. Hennessy, A. Kiehn. *Observing localities (Extended Abstract)* In: A. Tarlecki(ed.) Mathematical Foundations of Computer Science, LNCS 520, Springer-Verlag

[Br88] E. Brinksma. *Information Processing Systems - Open Systems Interconnection - LOTOS - A Formal Description Technique based upon the Temporal Ordering of Observational Behaviour* Draft International Standard ISO8807, 1988

[Di65] E. W. Dijkstra. *Cooperating Sequential Processes* Technical Report EWD-123, Technological University, Eindhoven, 1965

[FM91] G. L. Ferrari,U. Montanari. *The observational Algebra of Spatial Pomsets* In: J. C. Baeten,J. F. Groote (eds.)CONCUR'91. LNCS 527,Springer-Verlag

[Fr92] N. Francez. *Program Verification* Addison-Wesley, 1992

[Ho85] C. A. R. Hoare. *Communicating Sequential Processes* Prentice-Hall, 1985

[Mi85] G. J. Milne. *Circal and the Representation of Communication, Concurrency and Time* ACM Transactions on Programming Languages and Systems, Vol.7 pp270-298, 1985

[Mi89] R. Milner. *Communication and Concurrency* Prentice-Hall, 1989

[Old91] E. R. Olderog, *Nets, terms and formulas* Cambridge University Press, 1991.

[Pr87] K. V. S. Prasad. *Combinators and Bisimulation Proofs for Restartable Systems* PhD Thesis, Computer Science Department, University of Edinburgh, 1987

[Si92] A. Sinha. *Client-Server Computing* Communications of ACM, Vol.35 No.7;77-98, 1992.