On the Causal Description of Interactions in Physically Distributed Systems

Augusto Ciuffoletti
Dipartimento di Informatica
Universitá di Pisa

March 25, 1985

Abstract

The description of distributed activities is formally approached. A formalism is introduced for the description of cause-effect relationships among states on distinct components. Three orthogonal operators are defined to describe the evolution of physically distributed systems. An abstraction operator is defined to derive a description based on distributed activities from the description based on centralized components.

Only for purpose of "normalization", some categorial vocabulary has been used. For the complete comprehension of this report it is enough to know little more than the definition of category and functor.

The material contained in this report is subject to continued revision and update. Although carefully self contained, it is not complete, as its concern is a research in progress.

The reader interested in knowing about the development of this work may ask for updated documentation to the author at the following address:

Dipartimento di Informatica
Corso Italia N. 40
56100 PISA
ITALY

1 Introduction

The purpose of this report is to explore the field of the description of distributed computations.

This paper is an initial step through this topic: we first define a very general model to describe the interactions among the behaviors of several "open" components or entities. Each entity operates sequentially, and is considered to be physically centralized. The model admits the "action at distance" among different components, that is the asynchronous observation of external behaviors. Then we abstract the model in order to represent the behavior of distributed entities; for this new model to be significant, we have to restrict the range of possible behaviors of the centralized entities. We formally define a restriction which bounds the communication to be spatially pointwise (in a sense synchronous). Therefore, the restricted model puts some condition over the "action at distance".

More specifically, we proceed through three steps, each having an individual motivation.

The first step consists in the definition of a very general model for computations involving several sequential components whose operations have mutual side effects. The sequential components we consider are finite state machines. A transition leaves the process in a new state, that we say to be caused by all the states that were observed during the transition. When we speak of "observation" of a state, we refer to the fact that the process moves (or performs a transaction) depending on the information contained in the aforementioned state.

We also consider the operation of creating a new component dynamically.

An implementation of this model is then introduced, described as a data type containing the histories, with some operators over them. The exposition of the implementation has a categorial flavor.

The second step introduces a constraint on the observability of the states. Informally, we assert that only the present is observable; this corresponds to an hypothesis of strong synchronism. We show that such a behavior may be simulated with a far less strict constraint. Under these hypotheses, the description of an history may focus on the structured cooperation of distinct processes. Each instance of cooperating activity can be considered a distributed entity, causally related to other distributed entities. So a new cause- effect model can be defined, having as active object these distributed entities.

For both the two models, we give a running implementation. The implementation is more demonstrative then of practical utility; what we have done has been to describe in Prolog the formal definitions given above. This with the purpose of simulating the abstract concepts, to give them a practical evidence, and to give the basis of a prototyping tool, to test the power of the protocol and of the underlying model.

2 The basic model

The purpose of the basic model is to describe the cause- effect relation among the states of several interacting finite state machines.

We are not interested in the description of the internal states, but in the analysis of the cause-effect relations among the states; thus, we will have a very concise and implicit notation to describe the states, and a very analytical and explicit notation to describe the cause-effect relation. We assume a state being denoted by a function with argument the process and the local timestamp associated to the state.

Definition 2.1 (state) *:*

state : PROCESS X INTEGER$\Leftrightarrow$STATE*(see appendix)*

For what is concerning the temporal relation, we define a peculiar property, that, in general, characterizes the cause- effect relations: this property asserts that, to represent a cause-effect pattern, a generic relation must not contain cycles (of any length).

Definition 2.2 (cause-effect relation) *:*
A relation $\succ$ on the set A is cause-effect iff

$$\forall a \in A, \neg\exists x_1, ..., x_n \in A$$
$$a \succ x_1 and ... and x_n \succ a.$$

This property means that a state cannot cause itself through a sequence of cause-effect steps. This assertion formalizes the fact that the cause-effect sequence is time consuming.

A cause-effect relation holds between the sequentially ordered states of the same component; further, we allow a state to be causally related (second argument in the causes relation) to any "past" state.

Let us call *causes* the cause-effect relation over the STATEset. Depending on convenience, we will use a prefix or infix notation; as an example,

causes(state(P,3),state(Q,2))

means that a message sent by process P at local time 3 has been received by process Q at local time 2 (the send-receive scenario is here introduced only for the sake of exemplification) and is equivalent to:

state(P,3) causes state(Q,2)

Thus the domain of our model (which we call HISTORY) is a set of objects representing the structures (states,causes), where states is a set of states over the different components, and causes is the cause-effect relation.

Definition 2.3 (history) *:*
Let (STATE)SET *(see appendix) be the power set of* STATE, CAUSES*be the binary relations over* STATE.

history: (STATE)SET X CAUSES$\overleftrightarrow{\Rightarrow}$HISTORY*(see appendix)*
history(S,causes) is defined iff

causes is cause-effect on S and	*# No loops,*
if $state(P,n) \in S$ *and* $n \neq 0$,	*# Timestamps are*
then $state(P,n-1)$ *causes* $state(P,n)$.	*# progressive.*

We define a function over our domain; let us call it event.

The event function has three arguments:

- an history,
- a set of pairs, each composed by:
  - a process name
  - a set of states

and returns a new history.

This history will be derived from the one given as argument adding a new state to the process named in the function call, causally related to the states indicate in the second component of the pair.

As an example consider the following history h1:

```
(P,0)     (Q,0)
  |
(P,1)
```

(an history can be represented with an Hasse diagram)

which may be transformed by

```
event(h1, {
          ( Q, {state(Q,0),state(P,0)} ),
          ( P, {state(P,1)} )
          } )
```

into

```
(P,0)     (Q,0)
  |  \      |
(P,1)     (Q,1)
  |
(P,2)
```

Definition 2.4 (Event) *:*
Let us define the utility:

present: HISTORY X PROCESS∘ →STATE
present(history(S,causes),P) is defined iff state(P,0) $\in S$.
present(history(S,causes),P) := state(P,n)
where:
$state(P, n) \in S$ *and*
$\forall s' \in S$, $\neg$ *(state(P,n) causes s')*

event : HISTORY X (PROCESS X (STATE)SET)SET → HISTORY *(see appendix)*
$event(history(S, causes), (P_1, S_{s1}), ..., (P_k, S_{sk}))$:=
history(S $\cup$ *new-S, causes* $\cup$ *new-causes)*

where:

new-S	$= \bigcup_{i \in [1,k]}\{$	$state(P_i, n+1)\|$	# *The new history contains*
		$state(P_i, n) \in S,$	# *the states immediately*
		$state(P_i, n+1) \notin S\}.$	# *following the last state*
			# *of the involved processes.*
new-causes	$= \bigcup_{i \in [1,k]}\{$	$(a, state(Pi, n+1))\|$	# *The causes relation is*
		$state(P_i, n) \in S,$	# *extended with the*
		$state(P_i, n+1) \notin S$	# *connections between*
		$s = state(P_i, n+1),$	# *the newly added states*
		$a \in S_{s_i}\}.$	# *and their causes.*

We define two further operators, the first with an essentially theoretical interest, the second with some pragmatic interest. The first is the "start history" function, which creates an empty history.

Definition 2.5 (start-history) *:*

start-history : →HISTORY
start-history := history({},{})

The second which creates a new process:

Definition 2.6 (create-process) *:*

create-process : HISTORY X PROCESS∘ →HISTORY
create-process(history(S,causes),P) := history (S $\cup$ *{state(P,0)} , causes).*

We claim the "event" and "create-process" functions to be partial; in fact their definitions map the arguments in a pair (states,causes), and then apply the "history function to this pair. But we know that "history" is not total; as a consequence, also the "event" and "create-process" functions are partial. We now state three fundamental conclusions about the definition domains of the three operators.

Proposition 2.1 *:*
event(history(S,causes),{(P_1*,*S_{s1}*), ...,(*P_k*,*S_{sk}*)}) is defined iff*

history(S,causes) is defined and		# *of course,*
H1) $\forall i,\ 0 \leq i \leq k$		# *for every evolving process*
	$state(P_i, 0) \in S$ *and*	# *the process already exists,*
	$S_{s_i} \subset S.$	# *the causes already exist.*

Proposition 2.2 *:*
create-process(history(S,causes),P) is defined iff

$history(S, causes)$ is defined and
H2) $state(P, 0) \notin S$.

Proposition 2.3 *: $start()$ is defined.*

Sketch of the proofs: the skeleton of the tree proofs is similar to an induction step. The third proposition looks like the "basis" of an induction, and is trivially proven though its own definition. The other two contain among the hypotheses a kind of "induction hypotheses" concerning a precedent history (consider that the result of these function is itself an history). Starting from the fact that the previous situation is an history (with the appropriate properties), and given the definition of the function and the additional properties over the other arguments, it is easy to show that the result is itself an history.

These properties define a kind of "well-formedness" of the instances of application of the operators.

Definition 2.7 (history-TRANSITION) *: An application of the operators event,create-process is an history-TRANSITION iff its arguments satisfy the hypotheses of the previous propositions.*

The following lemma is straightforward:

Lemma 2.1 *: Any composition of history-TRANSITION's beginning with a start and satisfying properties H1 and H2 returns an history.*

The proof follows an inductive scheme. As an example of composition of history-TRANSITION's, let us see the following expression:

```
event(
    create-process(
        event(
            create-process(
                start(),
                P),
            {(P,{state(P,0)})}),
        ,Q),
    {(Q,{state(P,0),state(Q,0)}),(P,{state(P,1)})}
```

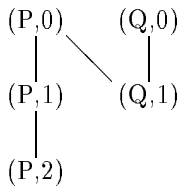
which is equivalent to the following sentence:

> "we start from an empty history and create process P; then process P autonomously moves; then process Q is created, and moves from its initial state getting some information from the initial state of P, while P autonomously moves"

and the history is the image of the pair

```
({ state(P,0),state(Q,0),state(P,1),state(Q,1),state(P,2)},
 { (state(P,0),state(P,1)),
   (state(P,1),state(P,2)),
   (state(Q,0),state(Q,1)),
   (state(P,0),state(Q,1)) }
```

which may be represented with the diagram:



Another important result is stated by the following:

Lemma 2.2 (no-junk property) *:*
*Any history is reachable through the composition of appropriate history-*TRANSITION*'s beginning with a start().*

which guarantees that, for any history, there exists a path to reach it from the empty history proceeding through well- formed "event" and "create-process" steps.

Sketch of the proof: the proof is by induction on the number of events in the history. The base of the induction seems to consist in proving that the empty history is an history, and is trivial. The inductive hypotheses asserts that any history with less than n states is reachable, and we must prove that any history of n states can be reached from one of less than n with well-formed steps. Such proof should follow the following trace:

- by absurd that there exists a $state(P, n)$ in the present (using the cause-effect hypotheses); (see previous definition of present)
- that the structure obtained removing that state and all the cause-effect relationships concerning it from the current history is an history itself;
- that the current history may be generated from the old one:
  - if $n = 0$ with a create;
  - else with an event having as causes all the s' s.t. *s' causes state(P,n).*

It is to be noted that the molteplicity of synchronous actions is not essential to the "no-junk" property and that the proof gives a procedure to translate any history in a path to reach it.

From the two lemmas it is now easy to derive the following conclusive theorem:

Theorem 2.1 *: $history(S, causes)$ is defined iff it is reachable through the composition of history-*TRANSITION*'s from the empty history.*

which gives an alternative, estentional definition of the domain of history.

2.1 Why just this model

This section contains some intuitive and pragmatic rationale for some of the choices operated during the design of the model.

First of all the decision to introduce the existence of sequential entities, namely the processes.

The aim of our model is to describe a distributed computation; this means that the activity is carried out through the cooperation of different entities, with some degree of persistence, each performing its operation in a centralized way. We assert that the time relations between the states of different processes are in general out of knowledge, unless derived from an explicit cause-effect relation. While the centralized nature of a single process allows the perfect knowledge of the order in which states occur. For these reasons we make of the process a key concept of our model, thus enforcing its basic nature. In fact, we assume that any distributed entity is composed (at some lower level of abstraction) of centralized entities.

Some important choices embedded in our model concern the cause-effect relationship.

We stress that a cause effect relation is not a partial order. As it influences some further theoretical work, this choice should be well motivated. Indeed, what we are trying to specify is the less constrained model to describe the cause-effect. It seems reasonable admit that a state cannot cause itself; a fortiori, it is consistent to require that this condition holds for arbitrary long sequences. The introduction of a transitive property would approach our model to an irreflexive partial order, also simplifying the specification of non-self-causality, but would also made less precise our model. Further, when necessary, the transitive property can be easely added (for example, in the non-self-causality property). This "non- necessity" excludes by itself the transitive property from our model; we will see in the next section how transitivity would drawback the definition of cooperation.

Lastly, we give a rationale to the propositions we have proved to subsist.

A rationale is in fact needed, since the hypotheses we have introduced in these properties may not be the only possible conditions to the closeness of the functions.

Let us begin with the event function. It is rather natural to suppose that the process involved in the event must be part of the system: this corresponds to say that its initial state is contained in the history.

A "by default" creation would be of little use, in some sense even confusing. It is also reasonable to suppose that the states that cause the new state are already present in the history; in fact it is hard to find an alternative. Less immediate is the fact that a process always evolves considering its present state; this choice can be justified noting that a "forgetful" interaction could be easely simulated with the creation of a new process and the halting of the stopped process (this could be done with an appropriate "forget process" function).

For what is concerning the hypothesis for the "create- process" to be defined, it seems rather reasonable that the new process has not yet been created.

2.2 Histories as a data-type

In this section we will give an implementation of the concepts described in the previous ones; in particular we will give a translation in terms of (elementary) category theory of the concept of history and of the related concepts.

For this purpose, we will introduce a very simple category, whose objects are the histories (that is, each history is an object of the category), and arrows are the events (or sequences of events). Let us call histories this category (the names labelling categories are bold).

We will denote the arrow with a name which is directly derived from the corresponding history-TRANSITION:

if h'= event(h",e) then event(e) $\in$ **histories**[h',h"].

Similarly, the

h' = create-process(h",P)

is represented with the arrow

create-process(P) $\in$ **histories**[h',h"]

This syntax is ambiguous, since the same string can denote two different arrows; due to the informal use of the notation, we prefer readability to uniqueness.

To have a category, we must define a composition operator over the arrows; the syntax of the composition is the following, according with a "programming language" style: i.e., the composition of the two arrows $a \in$ **histories**$[h, h']$ and $b \in$ **histories**$[h', h"]$ is written:

```
a,
b
```

(equivalent to the usual notation $b \circ a$); the composition is defined to be the sequential application of the functions associated to the arrows. E.g., given the two arrows:

```
create-process(P) ∈ histories[history({},{}),history({state(P,0)},{})],
event([[P,[[P,0]]]]) ∈ histories[ history({state(P,0),{}),
                                  history( {state(P,0),state(P,1)},
                                           {(state(P,0),state(P,1))}
                                           )
                                  ]
```

their composition will be denoted by the arrow

```
create-process(P),
event([[P,[[P,0]]])
```

belonging to

```
histories[ history({},{}),
           history({state(P,0),state(P,1)},
                   {(state(P,0),state(P,1))}
                   )
           ]
```

corresponding to the evaluation:

```
history({state(P,0),state(P,1)}, {(state(P,0),state(P,1))}) =
        event(
                create-process (
                                start()
                                ),P
                ),{(P,{state(P,0)})}
        )
```

We omit the proof that the composition we have defined is associative.

In the next section we shall describe an implementation of this concepts with the language Cprolog. The choice of this language is due to the fact that it is the most widely diffused language supporting a declarative programming style. This is useful to us for an easy (and reliable) translation of the formal specification into a running program.

2.3 The implementation

This section contains the description of the running implementation of the history data-type. The implementation has been written in Cprolog, and is presently running on the VAX 780 of the Department of Computer Science in Pisa.

We have implemented the three basic operators of the history data-type.

The implementation keeps a stack of the past histories of a system. When the program starts the execution, the stack is empty; any interacting request of performing a certain event (the syntax is described in the previous section) considers the top of the stack as the current history (derived according with the definitions) on the stack.

Let us see in deeper detail a sample session, derived from the example explained in the previous section. In response to the following clause:

```
:-start-history,
create-process(p),
event([[p,[[p,0]]]]),
create-process(q),
event([[q,[[p,0],[q,0]]],[p,[[p,1]]]]).
```

the system will build the following stack of histories:

```
history( [[[q,0],[p,0],[p,1],[q,1],[p,2]],
[[[p,0],[p,1]],[[p,0],[q,1]],[[q,0],[q,1]],[[p,1],[p,2]]]]).
history( [[[q,0],[p,0],[p,1]],
[[[p,0],[p,1]]]]).
history( [[[p,0],[p,1]],
[[[p,0],[p,1]]]]).
history( [[[p,0]],´[]´]).
history( [´[]´,´[]´]).
```

(this listing may be obtained with the query ":- listing(history).")

The bottom element `history([´[]´,´[]´])` has been generated after the first start-history of the "program"; the `create-process(p)` is an arrow from $history(\{\},\{\})$ to $history(\{state(p,0)\},\{\})$. Thus the new fact `history([[[p,0]],´[]´])` is pushed on the stack. Note that each fact has the form of the structure associated to an object of the histories category.

More precisely, the argument of history is a list of two elements:

a list of states: each state is a list of two elements, each representing a state, in the form of a list of two elements (a process and a timestamp),

a list of causes: (representing the causal relation); each cause is a list of two states (the cause and the effect).

The subsequent histories may be easely obtained applying the definitions given for the function event, start-history and create-process.

An annotated version of the program is given in the appendix (refer to pages with "basic" header).

3 Derived models

In the basic model, the presence of centralized entities (namely, the processes) is assumed. Instead, it may be of great interest to express the behavior of "physically distributed" entities. Informally, a physically distributed entity is determined by the "sum" of several centralized entities; for this sum to be meaningful, we must impose some sort of coordination among the centralized entities belonging to the centralized one.

In general, we characterize a certain kind of distributed entity (or activity, see /JEN80a/) according with the way in which the centralized entities are allowed to communicate among each other, to join or leave certain distributed entities; for instance:

- centralized entities (cents) may be part of one or more distributed ones (dents);
- cents can or cannot with other cents belonging to other dents;
- a cent can or cannot leave a dent;
- a cent can or cannot join a dent after its creation;
- and so on.

We note that the preceding list of possible choices is already able to characterize systems organized in atomic actions as well as CSP-like processes.

The choice of the specific model depends on the peculiar architectural needs.

As a general rule, to derive more specialized models, it is necessary to map the whole data type (that is histories and operators), in another, more appropriate data type, representing the acceptable behavior of the cents involved in dents.

To this purpose, it is necessary to define an appropriate subset of the histories, and restrict the operators in a suitable way. For what is concerning our meta language, we will call $< prefix >$-HISTORYthe new domain, where the prefix will in some sense recall the peculiar property of the elements of the subset. We shall try to use the same prefix also in the restricted operators.

We note that the restriction is equivalent to the specification of a subcategory.

The following step is to "abstract" the subcategory into a more concise structure, mapping the subsets of cents in "non- inspectable" distributed entities. This corresponds to the specification of a retraction from our subcategory into another category. We shall call the domain of this new category $< prefix >$-trace, with the forementioned meaning associated to the prefix.

In the following section we shall define a dent which represents the direct (that is "one step") cooperation of several entities in the determination of their next state. It is interesting to note that, to be able to describe the system in these terms, it is necessary to enforce a "quasi synchronism" property.

We will prove that this property may be preserved without the introduction of real time constraints and preserving a certain independence in the operation of the cents. More formally, we will prove that an heavily synchronous protocol is equivalent to a far less constrained one, that is, if an history is reachable in the first one, so is in the second, and viceversa.

4 The synchro trace model

This section describes a model derived from the basic one imposing a tight synchronism constraint. The next section will prove that such constraint is equivalent to a looser one. This result may be of interest as it guarantees that, using an appropriate non-synchronous protocol, it is possible to reach all and only those histories (or system states) reachable with a synchronous one. More explicitly, it indicates how to abstract synchronism from asynchronism.

The constraint we introduce to enforce a strong synchronism says that the only states readable during an event belong to the present of the system. To this purpose, we define the synchro-event operator, then we deduce from this the "synchro-reachable" subset of history, that is the SYNCHRO- HISTORY set.

We define the synchro-event function as an event with some further constraint binding its arguments:

Definition 4.1 (synchro-event) *:*

synchro-event : HISTORY X (PROCESS X (STATE)SET)SET $\circ \rightarrow$HISTORY
synchro-event(history(S,causes),$\{(P_1, S_{s1}), \ldots, (P_k, S_{sk})\}$*) is defined iff*
H3) $\forall i, 1 \leq i \leq k$, $\forall s \in S_{si}, \exists$ *process P, s=present(history(S,cause),P).*
synchro-event(h,e) := event(h,e).

We may define a synchro-TRANSITION, corresponding to the previous history-TRANSITION:

Definition 4.2 (synchro-TRANSITION) *: An application of the operators synchro-event, create- process is a synchro-*TRANSITION*iff its arguments satisfy the hypotheses H2 and H3.*

We may now define the SYNCHRO-HISTORYas the set of all the histories that may be reached through synchro-events.

Definition 4.3 (SYNCHRO-HISTORY) *:* SYNCHRO-HISTORY*is the subset of* HISTORY*containing the objects reachable through synchro-*TRANSITION*'s.*

The definition states that SYNCHRO-HISTORYrepresents the domain of all the computations performing interactions based exclusively on the present state.

It is interesting to stress that many problems posed by the design of distributed systems are greatly simplified under the hypotheses that the run-time support guarantees that at any time only the present of the system is considered to determine the future state; as an example, such a support guarantees against message shuffling or obsolete information utilization; it also makes easier the task of implementing backward recovery algorithms or atomic transactions.

Nevertheless, the definition is intensional, and does not identify the structure of the synchro-histories; to obtain an estentional definition of these objects, it is necessary to introduce some further formal concepts.

4.1 Formalizing the cooperation

The structure of the histories that can be reached through synchro-events has much to do with the concept of cooperation and, as a side effect, with atomic actions. Cooperation may seem not strictly bound to synchronism; indeed, the purpose of this section is that of giving a meaning to cooperation beyond the concept of synchronism.

Definition 4.4 (cooperates-with) *:*

cooperates-with : HISTORY$\rightarrow$STATE X STATE
cooperates-with(history(S,causes)) :=
$\{(s_1, s_2) \mid s_x,\ s_x \in S$ *and*
s_1 *causes* s_x *and*
s_2 *causes* s_x,
or
$s_x,\ s_x \in S$ *and*
s_1 *cooperates-with(history(S,causes))* s_x *and*
s_2 *cooperates-with(history(S,causes))* s_x $\}$

Intuitively, the cooperates-with relation binds two states having an effect in common and closes it up to transitivity. The relation is partial, in the sense that not every state is in at least one pair; more precisely, no state in the present does cooperate-with any other (remember the non-reflexivity of causes). Any state is in relation at least with itself, except present ones. We can conclude that, in the past of the system (past means "all the states that are not present), the cooperates-with relation is an equivalence relation.

We may imagine an equivalence class of cooperating states as a distributed entity composed by the processes whose the states belong to. The internal structure of such distributed entities is quite simple, being composed by a set of directly related states. We shall call this kind of dent "synchro-state". The synchro-states are related each other by a cause-effect relation, whose nature will be formally explored in the following of this report.

In conclusion, we are up to create a structure, similar to the history, but having the equivalence classes of cooperates-with instead of the states as entity.

For what is concerning the present states of the history (we recall that they do not belong to any equivalence class), we are going to treat them in a separate component of the structure.

Let us call SYNCHRO-TRACEa set of objects representing a structure composed by:

- the set of equivalence classes of cooperate- with, say the past of the system;
- a set of singletons, each containing a present state;
- a relation in the union of the previous sets.

More formally, we define a "quotient" function that, given an history, returns a synchro-trace.

Definition 4.5 (quotient) *: quotient :* HISTORY→SYNCHRO-TRACE

The definition of the function is this time split in three steps, each corresponding to a simpler function; that is:

quotient := synchro-trace ∘ f ∘ yrotsih.

The yrotsih maps an history in a pair (S,causes) and is the inverse of history.
The f is a function that associates to a pair (S,causes) a structure derived using the concept of cooperation.

Definition 4.6 (f) *: let* SYNCHRO-CAUSES*:* SYNCHRO-STATE X SYNCHRO-STATE *be the binary relations over* SYNCHRO-STATE

f: (STATE)SET X CAUSES→(SYNCHRO-STATE)SET X (SYNCHRO-STATE)SET X SYNCHRO-CAUSES
f(S,causes) := (Past,Present,synchro-causes)
where:
Past := { $t \mid \exists s \in t,\ x \in t$ *iff* x *cooperates-with(history(S,causes))* s}
Past is the set of cooperation classes
Present := { $\{s\} \mid \exists P,\ s =$ *present(history(S,causes))*}
Present is a set of singletons, each representing
a present state
synchro-causes := $\{(x, y) \mid (x \in$ *Past and* $y \in$ *Present* $\cup$ *Past) and*
two elements are in relation
if the first is in the past
and the second in the present or in the past and
$(\exists s_x \in x,\ \exists s_y \in y,\ s_x$ *causes* $s_y)\}$
there is a state in the first
and a state in the second that are causally related.

The synchro-trace function is a partial bijection on the SYNCHRO-TRACE's, which maps in this set only the triples (Past, Present, synchro-trace) generated starting from a SYNCHRO- HISTORY.

Definition 4.7 (synchro-trace) *:*

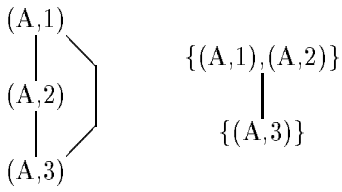
synchro-trace:(SYNCHRO-STATE)SET X (SYNCHRO-STATE)SET X SYNCHRO-CAUSES$\overset{\leftrightarrows}{\Rightarrow}$SYNCHRO-TRACE
synchro-trace(Past, Present, synchro-causes) is defined iff
$\exists h \in$ SYNCHRO-HISTORY, *(Past, Present, synchro-causes)* $= f \circ$ *yrotsih(h)*

The preceding definition of quotient is somewhat strictly aimed to map in the target set only the histories with a certain property. But this result is reached through a very implicit and indirect definition; the quotient function requires to be defined that the argument is a synchro-history, and the property of being a synchro-history is given by a sort of induction. As a first result of giving a more estentional definition of synchro-trace, we give the following:

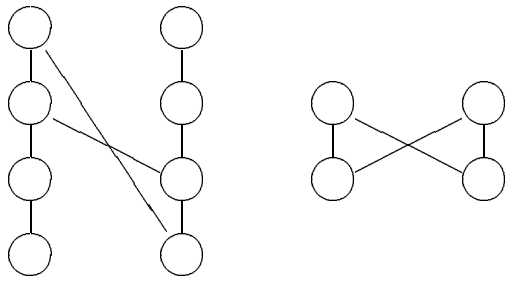
Theorem 4.1 *:*
Let $f \circ$ *yrotsih(h) = (Past,Present,synchro-causes)*
quotient(h) is defined iff synchro-history is cause-effect in Past U Present.

Sketch of the proof: the statement is equivalent to say that h is a synchro- history iff the corresponding synchro-causes is cause-effect. ($\Rightarrow$) h is a synchro-history; because of the generation rule for the synchro-histories, it is easy to prove that synchro-causes cannot have loops; then synchro-causes is cause-effect. ($\Leftarrow$) by induction on the number of states in history: synchro-causes is cause-effect; then, an element p of Present exists such that every p' synchro-caused by the element c of Past which synchro-caused p is in present (because of the no-loops property); then the state c together with p and all the p' define a synchro-event, to be applied to an history with less states and preserving the cause-effect property on the synchro- causes; the base of the induction is trivial.

Let us show with a counterexample that an history may have a quotient structure which is not cause-effect, that is which may not be a SYNCHRO-HISTORY:



Other histories not in SYNCHRO-HISTORY are the following:



as the quotient structures of both have cycles.

It seems, although we have no formal evidence of this, that any history not synchro contains one of the patterns exemplified in the previous examples and which may be labelled as:

- a message returns to the sender after the sender has moved;
- a message is received after another sent previously;
- a message is received from a process not yet modified by an already sent message.

The previous theorem gives an estentional definition of a synchro-history. Indeed, it says that the co-operation classes of such an object have a cause-effect have a cause-effect structure induced by the causes relation.

With this, we have also began to build the retraction from SYNCHRO-HISTORY (intended as a subcategory of HISTORY) and SYNCHRO-TRACE, a new, more abstract model.

Further, we have derived a structure strongly similar to the history, but which abstracts from the presence of processes; that is, starting from a structure expressing the local cause- effect relationship, and that this operation is possible for those histories reachable respecting a protocol in the interactions.

4.1.1 Cooperation without synchronization

Unfortunately, the protocol suggested by the synchro-event function implies an high degree of centralization, since any interaction has to be initiated and terminated simultaneously by all the participants (from this point of view see the "conversations" in /RAN78a/).

Thus we believe interesting to look for another definition of event which enables to reach all and only the synchro-histories, but which allows a lower degree of "real" synchronism. In other words, we are interested in a way to simulate synchronism starting from an asynchronous structure.

The following definition of structured-event solves this problem.

Definition 4.8 (structured-event) *:*

structured event: HISTORY X (PROCESS X (STATE)SET)SET $\circ \rightarrow$HISTORY
structured-event(history(S,causes),(P,S_s)) is defined iff
(let (Past,Present,synchro-causes) = f $\circ$ yrotsih (history(S,causes)))
$\forall s, t \in S_s$ *# every pair of classes whose*
(let s $\in$ class-s, t $\in$ class-t) *# elements are in the event*
then
$\neg\exists$ *class-s_1,...,class-s_n $\in$ Past* *# is not in the*
class-s synchro-causes class-s_1 and *# transitive*
... and *# closure of*
class-s_n synchro-causes class-t *# synchro-causes*
structured-event(h,e) := event(h,e)

As we have anticipated in the previous section , it holds that:

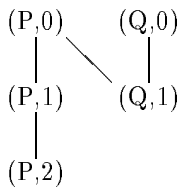
Theorem 4.2 *the history h $\in$* SYNCHRO-HISTORY *iff it is reachable through appropriate structured-TRANSITIONS (where the definition of structured-TRANSITION is similar to the previous synchro and history).*

The proof of this theorem is quite intricate and long: it basically consists in proving that this kind of event preserves the cause-effect property on the synchro-causes relation associated to any history. We plan to report about this proof in a separate report or in a specific appendix.

The theorem states that the synchro and the structured event have the same power. But the structured-event requires only one process to be concerned in a single event; because of this feature, we say that it is completely asynchronous. On the other hand, it requires some degree of organization in the interactions among the processes. From an implementative point of view, this will mean a certain degree of centralization and the existence of a "distributed protocol" to be respected. The necessity of such a coordination is formulated in the domain definition, allowing a bound access to past states. The protocol induced by this definition is less restrictive than the synchronous one; as an example, a processor can move after a send operation even if the corresponding receive has not yet been performed. This behavior was not allowed by the synchro- event.

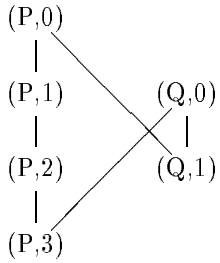
The step from synchronism to asynchronism cannot be by free. In fact, not every past state is readable. The "protocol" suggested by the definition of the domain of the structured- event binds the access to the states that are not mutually related through cooperation and causal relations.

As an example, a system having the history, already described in the previous sections:



does not allow the structured- event(h1,(P,state(P,2),state(Q,0))) since the cooperation classes state(P,0), state(Q,0) and state(P,2) are transitively related (through state(P,1)). It is fairly easy to observe that the same parameters would be unacceptable also for a synchro- event, while the unstructured event would accept them. The new history, obtained after such event could not be described in terms of cause-effect ordered cooperation classes;

in fact:



contains the cooperation class {(state(P,0), state(Q,0), state(P,2)} which reflexively precedes itself and determines a "perceivable delay" pattern.

As for the histories, a set of functions can be defined over the SYNCHRO-TRACE's, with a behavior equivalent to the structured-event, create-process and start. The meaning of this "equivalence" will be better explained in the next section, where a categorial view will be introduced.

The formal definition is not essential for our purposes, and is addressed to the specifically interested reader. For the concern of this report, it is enough to say that the new "transaction" function has, as arguments, the old synchro-trace and a set of set of SYNCHRO-STATE's, new objects similar to the STATE's in the history structure, but representing the cooperation classes in the new structure. Further considerations about these operators are contained in the section devoted to the implementation.

Definition 4.9 (transaction) *:*

SYNCHRO-TRACE X (SYNCHRO-STATE)SET) $\circ \rightarrow$SYNCHRO-TRACE
transaction(synchro-trace(Past,Present,synchro-causes),t) is defined iff
$t \subset Past \cup Present$ *and*
$\forall a, b \in t$,
$\neg\exists\ x_1, \ldots, x_k$,
a synchro-causes x_1 *and*
...
x_k *synchro-causes b.*
transaction(synchro-trace(Past,Present,synchro-causes),t):=
synchro-trace(Past',Present',synchro-causes').
where:
(let new-past $\in$ (SYNCHRO-STATE)SET*/(Past* $\cup$ *Present),*
new-present $\in$ (SYNCHRO-STATE)SET*/(Past* $\cup$ *Present))*
Past':=(Past/t) $\cup$ *{new-past} and*
Present':=(Present/t) $\cup$ *{new-present} and*
synchro-causes':=
synchro-causes /
$\{(x, y) \mid x \in t$ *or* $y \in t\} \cup$
{(x,new-past) | (x synchro-causes a and $a \in t$*)}*$\cup$
{(new-past,x) | (a synchro-causes x and $a \in t$*)}*$\cup$
{(new-past,new-present)}.

create : SYNCHRO-TRACE X (SYNCHRO-STATE)SET $\rightarrow$SYNCHRO-TRACE
create(synchro-trace(Past,Present,synchro-trace),t) is defined iff
$t \in Past \cup Present$.
create(synchro-trace(Past,Present,synchro-trace),t):=
synchro-trace(Past,Present $\cup$ *{t},synchro-trace).*

start : $\rightarrow$SYNCHRO-TRACE
start() := synchro-trace({},{},{}).

A strong relationships binds the operators on the histories and those on the synchro-traces. This relationships gives a constructive flavor to the theorem stated in this section, and allows to claim the existence of a functor from **histories**to **synchro-traces**.

Indeed, it is possible to define a correspondence between the structured events in the history structure and the transaction in the synchro-history. This correspondence preserves the composition and the identity in the two structures, i.e., if quotient is the name of the function from events to transactions, and e1,e2 are two generic structured- events, and eId, tId are the identities in the two structures, it holds that:

quotient(e1,e2) = quotient(e1,quotient(e2))

and

quotient(eId) = tId

This property should be proved. Indeed the proof seems to be more cumbersome than difficult; we hope that the implementation given in the appendix gives enough evidence to this claim.

4.2 Synchro-traces as a data type

We shall now present an implementation of the synchro- trace structure described in the previous section; again, the presentation will have a categorial flavor. We shall first define a subcategory of the histories and then we will give a functor transforming this new structure into the synchro-trace.

We will call the subcategory **synchro-histories**. The objects of synchro-histories are all the histories reachable through structured-TRANSITIONs. The arrows are the single events in this set, and their acceptable sequences. As proved in the previous section, this category has exactly the same objects as the one possibly generated by synchro- TRANSITIONs, but has more arrows.

The syntax will be similar to that defined for the history. Now we may define a functor from synchro-histories to synchro-traces.

The role of the functor will be that of abstracting from the presence of centralized entities (the processes), representing only the distributed entities (the equivalence classes of the "cooperate-with" relation). We will use the quotient function defined in the previous section to bind the objects of the synchro-history to the synchro-traces.

We note that this function is not one to one, since a certain synchro-trace corresponds to several non-isomorphic histories.

The next section will give better detail and some examples for what is concerning the behavior of the functor on the arrows.

4.3 The implementation

As for the history, here is the description of the implementation of the system trace in Cprolog.

The quotient function will also associate errors in the synchro-history to the corresponding histories in synchro- trace. Better detail about this function will be given in the next section.

The listing may be found in the appendix under header "synchro".

The implementation is divided in two parts: the first which implements the trace data-type, the second which maps the history in the trace data-type.

The implementation of the synchro-trace itself is quite similar to the history.

The following example should explain its main features. The following session:

```
:-start-synchro-trace,
create-state,
create-state,
transaction([a1]),
transaction([a2,a3]),
transaction([a4]).
```

brings to the following synchro-trace stack:

```
synchro-trace([[a7,a5],[a8,a6],[[a5,a6],[a5,a7],[a7,a8]]]). synchro-
trace([[a5],[a6,a4],[[a5,a4],[a5,a6]]]).
synchro-trace([[a3],[a4,a2],[[a3,a4]]]).
synchro-trace([´[]´,[a2,a1],´[]´]).
synchro-trace([´[]´,[a1],´[]´]).
synchro-trace([´[]´,´[]´,´[]´]).
```

The following transaction is not legal:

```
transaction([a5,a8]).
```

as `a5 synchro-causes a8` (see definition of transaction in 4.1.2). Instead the transaction:

```
transaction([a6,a8]).
```

pushes in the stack

```
synchro-trace([[a9,a7,a5],[a10],[[a5,a7],[a5,a9],[a7,a9],[a9,a10]]]).
```

Note that the program generates new names for the distributed entities created by the transaction.

For what is concerning the implementation of the "quotient" (see listing in the appendix, under header "quotient"). Being a functor from the history to the synchro-trace, the program may be split in two different functions: the first which transforms an history in a synchro-trace, the second that transforms an arrow in history in the corresponding arrow in synchro-trace (see note).

Note: the formal considerations say that the mapping is always defined only on synchro-histories, and for synchro- events, a subcategory of the histories. Instead, the implemented version accepts and tries to translate any syntactically correct event. The absence of "subtype checking" is justified by the prototypal intention of the work.

The former function builds, starting from the history on the top of the stack, the set of the cooperation classes; then it processes the causal relations between the states, in order to generate the causal relation among the classes. Finally, it assigns an arbitrary name to each class and builds a synchro- trace using these names as representative of synchro-states. The mapping from the classes to the synchro-states is kept in a data structure called "classes".

This synchro-trace is then pushed into a stack of synchro- traces.

The latter function accepts as parameter an event in the history data-type, and **using the existent "classes" data structure** generates the corresponding event in the synchro- trace data-type. Then it suitably updates the "classes" mapping and executes both the history and synchro-trace command. The two stacks are modified according with the two commands.

We have stressed that the implemntation uses an internal data structure to map directly the arrows of one category into the arrows of the other and applies them separately in the respective structures. The property given at the end of sect. 4.1.2 claims that the same result could be reached simply mapping the target of the arrow in the history category (say the resulting history after the application of the specified event) into the corresponding object in the synchro-trace category (say the resulting synchro-trace after the transformed event).

Our motivation to implement directly this transformation is threefold:

- the explicit implementation of the transformation gives constructivity to the theorem stated in sect. 4.1.2
- the transformation of the history with the quotient function generates a synchro-trace whose synchro-states have arbitrary names. Thus, for an algorithm which does not generate and execute an explicit synchro-trace arrow we generally have that the synchro-trace obtained applying the transformation of an history event will have different names for "homologous" synchro-states in the source synchro-history. This would drawback the readability of successions of synchro-traces.
- the consistent behavior of this procedure gives evidence to the claim that quotient is a function, which has not yet been proved.

We give in the pages under "tmp" header the complete listing of a session; the first six lines of the listing contain the initial prompt of the prolog interpreter and the precompiler messages. Then the listing of a program is given. In the first part of the program only the basic model is used; then, after the request of a quotient operation (which pushes in the stack the synchro-history corresponding to the current history) the computation is simulated on both models, applying the quotient operator on the following structured-events.

The complete listing of the stacks is then given: note that the classes and synchro-trace stacks contain only the elements generated after the quotient request. The interested reader is invited to draw the Hasse diagrams corresponding to the evolving structures.

Finally an approximation of the elapsed cpu time is given (consultations included).

A References

/JEN80a/ E.D. Jensen, "Distributed Systems", Advanced Course on Distributed Systems Architectures and Implementation, Lecture Notes in Computer Science, Springer-Verlag, 1980.

/RAN78a/ B. Randell, P.A. Lee, P.C. Treleaven, "Reliable Issues in Computing System Design", Computing Surveys, Vol. 10, No. 2, June 1978, pp.123-165.

B Notation

(SETNAME)SET : denotes the power set (or set of all the subsets) of the set *setname* (e.g. (STATE)SET is the power set of STATE).

$\Leftrightarrow$: denotes a bijection.

$\Rightarrow\!\!\!\!\leftarrow$: denotes a bijection defined on a subset of the target (i.e. the inverse is partial).

$\rightarrow\!\!\!\!\leftarrow$: denotes a function defined on a subset of the domain (i.e. is partial).

$\circ\!\rightarrow$: denotes a partial function.

$\rightarrow$: denotes a function.

C Syntax of the function definitions.

The definition of a function is always split in:

Type definition which defines the domain and the target, together with some basic features of the function (total or partial, existence of the inverse ...);

Domain definition (optional) if the function is partial, defines its domain.

Function definition (optional) if the function is not a bijection, intensionally defines the mapping from the elements of the domain to those in the target.