SPECIFICATION LANGUAGES FOR DISTRIBUTED SYSTEMS

by Pierpaolo Degano and Ugo Montanari

Dipartimento di Informatica, University of Pisa

Corso Italia 40, I-56100 Pisa, Italy

Abstract

Requirements of specification languages for distributed systems are considered, and a two level approach based on a kernel metalanguage and many application-oriented extensions is advocated. The method is applied to some models developed by the authors, organized in a tree-like refinement structure.

1. Introduction

Emphasis on the specification phase within the software life cycle has been suggested as a remedy against the many inconveniencies of presently available programming methodology. A more structured approach and a complete documentation of the design decisions taken in all the phases from requirements to coding should enable an easy modification of the resulting software product for maintenance and re-use. It has been also suggested that the main loop of the software life cycle be closed on an executable version of the program, completely specified but still not optimized, rather similar to a detailed specification /BCG83/. Fully optimized versions should be derivable either manually or semiautomatically but should not be the "main documents" of the software product.

Improvements on programming methodology in the specification phase demand suitable specification languages. They should be formally defined, executable, and easy to use. In this field close collaboration between computer scientists on one side and software engineers on the other is badly needed. The situation is even more demanding in the case of concurrent distributed systems, where many theoretical problems are still open and where our intuition is often inadequate. A number of recent workshops /Spec79, 83, 84/ are evidence of the interest in specification languages for sequential and concurrent systems.

This work has been partially supported by CNR - Progetto Finalizzato Informatica, Obiettivo Cnet.

2. Requirements for a specification language.

In a specification language we can distinguish two aspects. The first is related to its semantic definition and the second refers to its flexiblity and usability in a number of practical situations. The two aspects are somewhat conflicting, since the former leads to elementary and orthogonal constructs, while the latter tends to require the contemporary presence of many, overlapping constructs and languages, specialized for levels of abstraction and fields of application.

From a practical point of view, the presence of several languages in the specification of a large system will probably be a fact of life and it will be necessary to cope with it. In principle, it might be enough to formalise all the specification languages using "the" mathematical language. However, this approach might lead to serious practical problems, since if the specification languages employed can be interfaced only at a very primitive level (e.g. set theory) it will not be easy to reason and prove properties about the programs so specified. Furthermore, it will be almost impossible to efficiently implement all the languages used to specify the system in order to obtain early prototypes.

The above conflict can be solved by means of a two-level approach. First,a basic metalanguage should be introduced containing all the concepts needed, organized in an elementary way but well studied and easy to understand; second, several specification languages should be defined in terms of the metalanguage using few, simple extension mechanisms. A specification might become executable by providing an implementation of the metalanguage and interpreters (translators) for every language towards the metalanguage.

A good example of the above two-level approach is the Pebble specification language /BuLa84/, where the kernel is a version of typed lambda calculus and the mechanisms for modularization and data abstractions are provided as applications. The semantics of Pebble is operational, and is given in the style of SOS /Plot83/ using labelled transition systems.

Since we are interested in specification languages for concurrent, distributed systems, more concepts must be embedded in the metalanguage. Necessary constructs include at least primitives for describing nondeterminism, concurrency, synchronization and communication. An important step towards a metalanguage for concurrency is Milner's asynchronous and synchronous CCS /Miln80, 83/. It has rather few constructs, a rich and elegant theory and an operational semantics defined in terms of labelled transition systems.

A construct for parallel composition is available in CCS. However, its meaning can be expressed in terms of other operations, and thus it is not primitive in this sense. In fact, CCS, as well as all the models based on interleaving, describes the fact that a set of events may occur concurrently (independently from each other) by saying that they may occur in any order. In this way a total ordering among the possibly spatially separated and causally independent events is imposed.

Although this level of detail is adequate for many applications, according to a number of researchers it is insufficient to describe all those aspects of distributed, concurrent systems that have practical interest, e.g. fairness. Models have been proposed which use partial orderings to explicitely describe the fact that events may take place concurrently. Among these we mention the pioneering work on Petri nets /Bram83/ and Cosy by Lauer et al. /LTS79/. Also the authors have followed this approach defining models which will be used in the sequel as a case study.

A major motivation for formally specifying a system is to be able to prove properties about it. Providing a satisfactory proof system is not a simple matter, especially for an operationally defined metalanguage. A widely proposed approach takes a temporal logic as a starting point /Roev84/.

Once a satisfactory metalanguage has been designed, among the most needed extensions we mention those providing the ability of structuring and composing pieces of specifications. The features required concern parametrization, modularity and abstraction.

The use of a specification language for a sizable system is greatly improved by the availability of suitable tools. Many of the considerations valid for standard programming environments also apply to specification language tools. In particular, the syntax- or semantic-driven techniques used for adapting generic tools (like editors, type checkers, interpreters, debuggers, etc.) to a particular language, should be more convenient in a specification language environment, due to the hopefully simpler structure of specification languages themselves.

We already mentioned the convenience for a specification language to be executable in a reasonably efficient way. Of course the use of high parallelism and/or special purpose machine architectures may be of great help. However, we believe that the practical possibility of executing specifications, e.g. for early prototyping, must be a specific concern in designing them, since otherwise the inefficiencies can easily, and hopelessly, increase exponentially.

Finally, we mention the so-called human engineering issue. If a specification language has to be used at all in practice, it must require only a limited knowledge by the programmer of the deep theoretical issues involved in its definition. It must also be intuitively appealing and should use all the technically available expedients (e.g. sophisticated graphics) for achieving an easy interaction with the user (see for instance the documents of the ESPRIT project Graspin /GRAS84/).

3. A basic model and its refinements

In this section we follow the methodological guidelines surveyed in the previous section by presenting in some detail several models developed by the authors. All formalisms consider concurrency a basic, irreducible concept and are based on partial orderings. The models are organized in a tree-like refinement structure, starting from a basic model and adding to it independent features to describe different aspects of distributed concurrent systems. These features are meant as a kernel of a more elaborated specification language, which should combine them in a usable way.

3.1. Concurrent histories

In this section we introduce our basic model, which is intended as Level 1. of our structured presentation.

We give an informal introduction to the notion of concurrent history and to the semantics of a set Z of atomic histories. A detailed definition can be found in /DeMo84a, b/.

Let A be a countable set of observable actions and let E be a countable set of process types containing an element 0 called termination. Sets A and E are disjoint.

A concurrent history h in H_{fin} is a triple $h=(S,l,\leq)$ where:

S is a finite set of subsystems;

l is a labelling function

$l:S \rightarrow A \cup E$ and

$\leq$ is a partial ordering relation on S.

The subsystems with labels in A are called <u>events</u>, while those with labels in E are called process states or simply <u>processes</u>.

We require that events never be minimal nor maximal, the processes always be minimal or maximal, but not both, elements of $\leq$. Processes which are minimal are called <u>heads</u>, and processes which are maximal are called <u>tails</u>. Thus processes are partitioned into heads and tails. Two histories are called <u>disjoint</u> if their sets of subsystems are disjoint.

Two histories $h_1=(S_1,l_1,\leq_1)$ and $h_2=(S_2,l_2,\leq_2)$ are <u>isomorphic</u> iff there is a bijective mapping

$g:S_1 \rightarrow S_2$ such that

$l_1(s)=l_2(g(s))$ and

$s_1\leq_1 s_2$ iff $g(s_1)\leq_2 g(s_2)$.

We define an associative nondeterministic <u>partial</u> <u>replacement</u> operation on histories. Given two disjoint histories h_1, h_2 and a history h, we write h_1 <u>before</u> h_2 <u>gives</u> h iff h can be obtained by the following procedure. A possibly empty subset S^h_2 of the head processes of h_2 is matched against a subset S^t_1 of the tail processes of h_1, and corresponding processes are identified. Of course two processes can match only if their labels are identical. If the subset S^h_2 is the whole set of the head processes of S_2, the operation is called <u>full replacement</u> or simply <u>replacement</u>, and set S^t_1 is called <u>rewritable</u>.

The relation $\leq_1 U \leq_2$ is then made transitively closed and the processes in the matching set $S^h_2=S^t_1$ are erased. Note that we have

$S=(S_1-S^h_1)U(S_2-S^t_2)$.

In Fig. 1 we see an example of replacement. Here a,b,c are in A, and E_1, E_2 are in E. Partial orderings are depicted through their Hasse diagrams, growing downwards. Processes (events) are represented as boxes (circles).

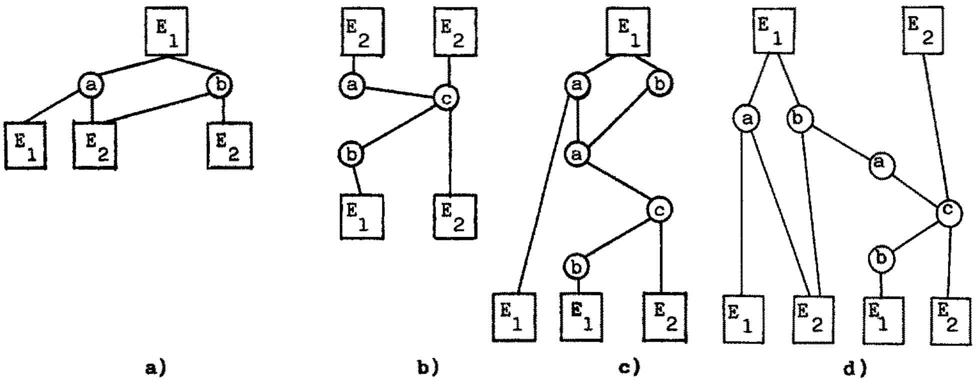


Fig. 1. Four concurrent histories h_1 (in a)), h_2 (in b)), h_3 (in c)) and h_4 (in d)) such that h_1 before h_2 gives h_3, and h_1 before h_2 gives h_4.

A history is atomic if either

i) there are no events and each head is smaller in the partial order than all the tails; or

ii) there is exactly one event greater than all heads and smaller than all tails.

An atomic history represents a single synchronization, either unobservable or observable.

We introduce a linear representation (up to isomorphism) for atomic histories:

i) $M_1 \dashrightarrow M_2$

ii) $M_1 \overset{a}{\dashrightarrow} M_2$

where M_1, M_2 are multisets of process types and a is an observable action.

Let Z be a set of disjoint histories.

A computation on Z is a finite or infinite sequence $D= \{h_i\} =(h_0,h_1 \ldots)$ such that

h_i before r_i gives h_{i+1} $\quad i=0,1,\ldots$

where h_0, r_i $(i=0,1,\ldots)$ are disjoint atomic histories isomorphic to histories in Z.

As an example consider a computation with

$h_0 = E_1 \longrightarrow E_3, E_4$

$r_0 = E_3 \xrightarrow{a} E_1, E_5$

$r_1 = E_4 \xrightarrow{b} E_6, E_2$

$r_2 = E_5, E_6 \longrightarrow E_2$

where h_3 is the history in Fig 1a). Notice that all replacements are full. Note also that histories r_0 and r_1 might be interchanged obtaining the same h_3.

The histories belonging to a computation on Z are called derivable from Z. Furthermore, the result of a finite computation is its last element, if it does contain no rewritable set. The result of an infinite computation is its limit, if any, in a suitable metric space. Actually, in /DeMo84a/ (where replacement is required to be full) four distances on finite histories are defined, and the limits are obtained through standard topological completions. Remarkably enough, the non terminating computations converging in the four resulting complete metric spaces enjoy interesting liveness properties.

The four properties are: vitality (every running process will eventually produce an observable event), global fairness (a synchronizable set of processes will eventually run), local fairness (a process which is repeatedly ready to run, possibly with different partners, will eventually run), partial deadlock freedom (every non-terminated process will eventually run).

Moreover, the limits in the metric spaces are directly characterized in terms of their, structural properties.

The proposed approach proves to be fruitful: a nondeterministic universal scheduler is defined which is capable of generating all and only computations being convergent in a given metric. This scheduler can be used in the four cases above, thus keeping only those computations which have the desired liveness property. The metric is a parameter of the scheduler, which is thus independent of the particular liveness property under consideration.

Our notion of concurrent history expresses in an abstract fashion the idea of concurrent computation. In fact, events not related by the partial ordering $\leq$ are meant to be concurrent. On the other hand, the above given notion of computation is purely sequential, and, since every computation step generates at most one event, a computation induces a well-founded total ordering on the events of its last element or of its result. We call it generation ordering.

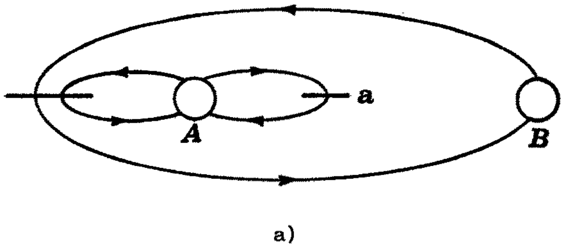
The following theorem given in /DeMo84b/ links the sequential and concurrent notions of computation.

Theorem 3.1 The generation orderings induced by all computations on Z having the same history $h=(S,l,\leq)$ as last element or as result, are exactly those well-founded total orderings compatible with (i.e. larger than or equal to, in the set theoretical sense) the partial ordering $\leq$ of h.

In /DeMo84b/ a construction is presented for generating a most simplified Labelled Event Structure (LES) starting from a set Z of atomic histories. This LES defines the semantics of Z. Labelled Event Structures are models of nondeterministic concurrent computations described in the literature /CFM82/, /Wins82/. Our notion of simplification is based on an abstraction homomorphism having the property that, given a LES, a unique most simplified homomorphic LES always exists.

3.2. Petri nets

The notion of computation on a set Z of atomic histories defined in the previous section is a general language-independent framework for describing behaviours of concurrent programs. The set Z can be defined independently. If Z is finite, it can be given explicitely. We call this Level 1.1.. In this case a set Z is equivalent to a transition Petri net /JaVa79/ plus a partial function mapping the transitions of the net into observable actions. The places of the net correspond to the process types, while every transition is associated to an atomic history in Z. More precisely, immediate antecedent (successor) places of the transition correspond to heads (tails), and an event exists iff the partial function above is defined. In fig. 2 we see an example of this equivalence.



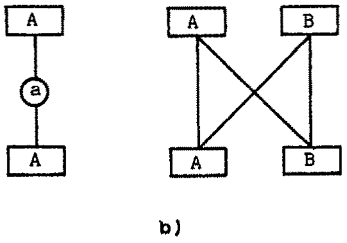


Fig. 2 a) A transition Petri net. b) The equivalent set of atomic histories.

3.3. Synchronization

So far, an atomic history involving many processes is understood as a single, indivisible move. On the other hand, it can often be conveniently seen as the composition, through a synchronization mechanism, of a separate move for each process involved. This consideration brings us to Level 1.2.. For the sake of symplicity, from now on we consider only atomic histories of type ii). Each possible move is described by a labelled production. The left member of a production contains a single process, and the label is called a communication protocol. The synchronization mechanism is represented by an associative commutative partial function f, mapping, if defined, any multiset of communication protocols into an observable action. Function f, called synchronization function, makes it possible to specify which productions can be composed in a single transition rule.

A production is a triple

$A \xrightarrow{p} M$

where A is a process type, M is a multiset of process types and p is a protocol.

Given a set W of productions and a synchronization function f, the atomic histories in Z are derivable by the following inference rule

$$\frac{A_1 \xrightarrow{p_1} M_1, \ldots, A_n \xrightarrow{p_n} M_n}{A_1, \ldots, A_n \xrightarrow{f(p_1, \ldots, p_n)} M_1 \cup \ldots \cup M_n}$$

In Fig. 3 we see a computation step where the atomic history used is generated by the above rule.

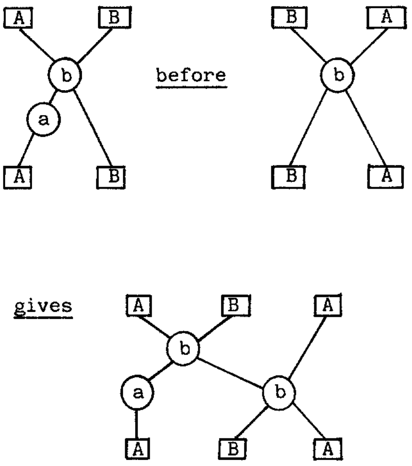
An example of a synchronization mechanism described in the literature which fits our present schema is that used for synchronization trees /Miln80/. Milner, however, does not distinguish between observable actions and communication protocols, since the observer himself is considered to be a process. Furthermore, the communication mechanism is intended to be based on message passing, and thus actions come in pairs like $a\bar{a}$, where an element represents the envoy of the message and the other its reception.

$A \xrightarrow{a'} A$ $\qquad$ $f(a')=a$ $\qquad$ $\dfrac{A \xrightarrow{b'} A \;,\; B \xrightarrow{b''} B}{AB \xrightarrow{b} AB}$

$A \xrightarrow{b'} A$ $\qquad$ $f(b',b'')=b$

$B \xrightarrow{b''} B$

a) $\qquad$ **b)** $\qquad$ **c)**



d)

Fig. 3 a) Three productions (notice the half-arrow).

b) The synchronization function.

c) The derivation of an atomic history.

d) A step in a computation.

To adapt synchronization trees to our framework, we assume that the observer can exchange a set of protocols $a'_1, \bar{a}'_1, a'_2, \bar{a}'_2, \ldots$. Thus we have,

$f(a'_1)=a_1,\ f(\bar{a}'_1)=\bar{a}_1, \ldots$

$f(a'_1,\bar{a}'_1)=\tau,\ f(a'_2,\bar{a}'_2)=\tau, \ldots$

where τ represents Milner's unobservable action.

Conversely, the "intersection" synchronization mechanism proposed by Hoare /BHR84/ does not fit completely. This mechanism forces all processes which are present in the system to agree on the same protocol, i.e.

$f_n(a',\dots,a')=a,\ f_n(b',\dots,b')=b,\dots$

where the synchronization function f_n applies iff there are exactly n processes in the system. Notice that this synchronization mechanism requires access to the global state of the system, and we exclude this possibility.

3.4. CCS

In the previous section we were able to define our set Z of atomic histories in terms of a given set of productions and a synchronization function. It is also possible to define directly Z using suitable inference rules, obtaining Level 1.3.. Here we define Z_{CCS} for Milner's CCS. It is possible to prove that there is a direct correspondence between atomic histories in Z_{CCS} and CCS derivation steps (/DDM84/). Recall that the concrete syntax of pure CCS terms is as follows.

$E::=x \mid NIL \mid \mu E \mid E\backslash\alpha \mid E[\phi] \mid E+E \mid E|E \mid rec\ x.E.$

We introduce a notion of move

$$I_1 \overset{L}{\underset{I_3}{\longrightarrow}} I_2$$

which generalizes Milner derivation relation

$$E_1 \xrightarrow{\mu} E_2$$

and which is essentially an atomic history, according to the definition given in the Section 3.1. The elements I_1, I_2 and I_3 in a move are finite sets of grapes, i.e. of terms defined as follows.

$G::= E \mid id|G \mid G|id \mid G\backslash\alpha \mid G\,[\phi]$

where E denotes a CCS term and α and $[\phi]$ have the same meaning as in E.

Intuitively speaking, a grape represents a suitable subterm of a CCS term, together with its access path. Given a move $I_1 \overset{L}{\underset{I_3}{\longrightarrow}} I_2$, the grapes in I_1, I_2, I_3 are called head, tail and idle grapes, respectively.

A CCS term can be decomposed into a set of grapes by the function dec, here defined by structural induction.

dec(NIL) = $\{$NIL$\}$

dec(x) = $\{$x$\}$

dec(μE) = $\{\mu$E$\}$

dec(E$\backslash\alpha$) = dec(E) $\backslash\alpha$

dec(E[ϕ|) = dec(E)[ϕ]

dec(E_1+E_2) = $\{E_1+E_2\}$

dec($E_1|E_2$) = dec(E_1)|id U id|dec(E_2)

dec(rec x.E) = $\{$rec x.E$\}$

Here the application of a syntactic constructor to a set of grapes is defined as applying the constructor to all grapes in the set, e.g.

I$\backslash\alpha$ = $\{$G $\backslash\alpha$ | G in I$\}$.

Notice that the decomposition stops when an action, a sum or a recursion in encountered. We have for instance:

(3.1) dec((((rec x.αx+βx)|rec x.αx+γx)|rec x.$\bar{\alpha}$x)$\backslash\alpha$)=

$\{$(((rec x.αx+βx)|id)|id)$\backslash\alpha$, ((id|rec x.αx+γx)|id)$\backslash\alpha$, (id|rec x.$\bar{\alpha}$x)$\backslash\alpha\}$.

It is easy to see that function dec is injective, and thus full information about E is contained in dec(E). It is not surjective instead, and a set of grapes which is the decomposition of a CCS term is called <u>complete</u>.

Going back to our definition, the last element of the move relation is a <u>synchronization term</u> L, namely a term defined as follows.

L:= μ | id|L | L|id | L|L | L$\backslash\alpha$ | L[ϕ]

where μ,α and ϕ have the same meaning as in CCS terms.

Intuitively, L brings information about the internal communication of the move. We define by structural induction on L a function syn, which embodies the synchronization algebra of CCS.

syn(μ) = μ

syn(id|L) = syn(L|id) = syn(L)

syn($L_1|L_2$) = <u>if</u> syn(L_1) = $\overline{\text{syn}(L_2)}$ <u>then</u> τ <u>else</u> synerror

syn(L $\backslash\alpha$) = <u>if</u> syn(L)=α <u>or</u> syn(L)=$\bar{\alpha}$ <u>then</u> synerror <u>else</u> syn(L)

syn(L[ϕ]) = <u>if</u> syn(L)=synerror <u>then</u> synerror <u>else</u> ϕ(syn(L))

Notice that if a subterm of L evaluates to synerror, the whole L evaluates to synerror.

We are now ready to define our move relation using axioms and inference rules in direct correspondence with those of Milner derivation relation.

Let

$$I_1 \xrightarrow[I_2]{L} I_3$$

be the least relation which satisfies

<u>Act</u>. $\{\mu E\} \xrightarrow[\emptyset]{\mu} \mathrm{dec}(E)$

<u>Res</u>. $I_1 \xrightarrow[I_3]{L} I_2$ <u>implies</u> $I_1\backslash\alpha \xrightarrow[I_3\backslash\alpha]{L\backslash\alpha} I_2\backslash\alpha$

<u>Rel</u>. $I_1 \xrightarrow[I_3]{L} I_2$ <u>implies</u> $I_1[\phi] \xrightarrow[I_3[\phi]]{L[\phi]} I_2[\phi]$

<u>Sum</u>.1) $I_1 \xrightarrow[I_3]{L} I_2$ <u>implies</u> $\{E_1+E\} \xrightarrow[\emptyset]{L} I_2 \cup I_3$

2) <u>and</u> $\{E+E_1\} \xrightarrow[\emptyset]{L} I_2 \cup I_3$

<u>where</u> $\mathrm{dec}(E_1)=I_1 \cup I_3$

<u>Com</u>.1) $I_1 \xrightarrow[I_3]{L} I_2$ <u>implies</u> $I_1|id \xrightarrow[I_3|id \cup id|\mathrm{dec}(E)]{L} I_2|id$

2) <u>and</u> $id|I_1 \xrightarrow[id|I_3 \cup \mathrm{dec}(E)|id]{L} id|I_2$

3) $I_1 \xrightarrow[I_3]{L} I_2$ <u>and</u>

$I'_1 \xrightarrow[I'_3]{L'} I'_2$ <u>implies</u> $I_1|id \cup id|I'_1 \xrightarrow[I_3|id \cup id|I'_3]{L|L'} I_2|id \cup id|I'_2$

<u>Rec</u>. $I_1 \xrightarrow[I_3]{L} I_2$ <u>and</u>

$I_1 \cup I_3 = \mathrm{dec}(E_1[\mathrm{rec}\ x.E_1/x])$ <u>implies</u> $\{\mathrm{rec}\ x.E_1\} \xrightarrow[\emptyset]{L} I_2 \cup I_3$

The intuitive meaning of the move relation

$$I_1 \xrightarrow[I_3]{L} I_2$$

is that the grapes in I_1 become the grapes in I_2 with internal communication L, while the grapes in I_3 stay idle. It is easy to see by induction that $I_1 \cup I_3$ and $I_2 \cup I_3$ are complete sets of grapes.

Therefore, given a move we can syntetize out of it a *head* term E_1 and a *tail* term E_2 such that $dec(E_1) = I_1 \cup I_3$ and $dec(E_2) = I_2 \cup I_3$. We can now shortly comment about our axiom and rules.

In axiom *Act*., a single grape is rewritten as a set of grapes, since the firing of the action makes explicit the (possible) parallelism of E. A move generated by rule *Sum*.1) can be understood as consisting of two steps. Starting from the singleton $\{E_1+E\}$ a first step discards alternative E and decomposes E_1 into $I_1 \cup I_3$; a second step (the condition of the inference rule) rewrites I_1 as I_2 and leaves I_3 idle. The net effect of the two steps, however, is to rewrite the singleton $\{E_1+E\}$ into the set $I_2 \cup I_3$, with no idle grape.

Rule *Com*.1) (*Com*.2)) can be read as follows. If we have a move where I_1 is rewritten as I_2 and I_3 stays idle, we can add in parallel to I_1, I_2 and I_3 to the right (to the left), a complete set of grapes dec(E), which stay idle. *Com*.3) is the synchronization rule: note that encoding the composition of L and L' into L|L' may permit to use different synchronization algebras.

As an example we show the proof of both a move and the corresponding Milner derivation.

$\{\alpha E'\} \xrightarrow[\emptyset]{\alpha} \{E'\}$, where E'= rec x.αx+βx, by *Act*.;

$\{\alpha E'+\beta E'\} \xrightarrow[\emptyset]{\alpha} \{E'\}$, by *Sum*. 1);

$\{E'\} \xrightarrow[\emptyset]{\alpha} \{E'\}$, by *Rec*.;

i) $\{E'|id\} \xrightarrow[\{id|E''\}]{\alpha|id} \{E'|id\}$, where E"= rec x.αx+γx, by *Com*.1);

$\{E\} \xrightarrow[\emptyset]{\bar{\alpha}} \{E\}$, where E=rec x.$\bar{\alpha}$x, by *Act*. and *Rec*.;

$\{(E'|id)|id,\ id|E\} \xrightarrow[\{(id|E'')|id\}]{(\alpha|id)|\bar{\alpha}} \{(E'|id)|id,\ id|E\}$, by i) and *Com*.3)

(3.2) $\{((E'|id)|id)\backslash\alpha, (id|E)\backslash\alpha\} \xrightarrow[\{((id|E'')|id)\backslash\alpha\}]{((\alpha|id)|\bar{\alpha})\backslash\alpha} \{((E'|id)|id)\backslash\alpha, (id|E)\backslash\alpha\}$.

Let us follow now the corresponding derivation in pure CCS.

$\alpha E' \xrightarrow{\alpha} E'$ where $E'=\text{rec } x.\alpha x+\beta x$

$\alpha E'+\beta E' \xrightarrow{\alpha} E'$

$E' \xrightarrow{\alpha} E'$

i) $E'|E'' \xrightarrow{\alpha} E'|E''$ where $E''=\text{rec } x.\alpha x+\gamma x$

$E \xrightarrow{\bar{\alpha}} E$ where $E=\text{rec } x.\bar{\alpha}x$

$(E'|E'')|E \xrightarrow{\tau} (E'|E'')|E$

$((E'|E'')|E)\backslash\alpha \xrightarrow{\tau} ((E'|E'')|E)\backslash\alpha$.

Note that if we write (3.2) as $I_1 \xrightarrow[I_3]{L} I_2$ and the latter derivation as $E_1 \xrightarrow{\tau} E_2$ we have $dec(E_1)=dec(E_2)=I_1 \cup I_3=I_2 \cup I_3$ by Example 3.1 and, according to the definition, $syn(((\alpha|id)|\bar{\alpha})\backslash\alpha)=\tau$.

In general it is possible to prove /DDM84/ that if $E_1 \xrightarrow{a} E_2$ then $I_1 \xrightarrow[I_3]{L} I_2$ with $syn(L)=a$, $dec(E_1)=I_1 \cup I_3$, $dec(E_2)=I_2 \cup I_3$, and viceversa.

We now translate moves to histories. Given a move

$$m = I_1 \xrightarrow[I_3]{L} I_2$$

with $syn(L)\neq synerror$, its head (tail) grapes in $I_1(I_2)$ can be interpreted as head (tail) processes of an atomic history, having one event labelled by $syn(L)$. Idle grapes are simply forgotten. However, since the sets of head and tail processes of a history are by definition non intersecting, while I_1 and I_2 may have non empty intersection, it is necessary to make "new" copies of I_1 and I_2. The same construction takes care of the fact that all histories in a set Z of atomic histories must be disjoint.

In Fig. 4 we see a picture of the history corresponding to the move in (3.2).

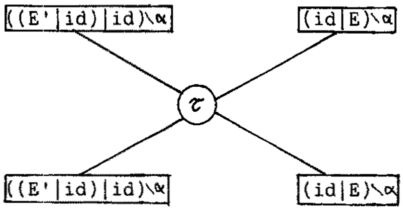


Fig.4 A picture of the history corresponding to the move (3.2).

3.5. Spatial distribution

In this section we describe Level 1.2.1., thus refining Level 1.2. of Section 3.3.. We enrich a history with a spatial structure by introducing a set of ports N and a function c specifying for every subsystem s the tuple of ports to which s is connected.

Formally, a distributed history is thus a quintuple $(N,S,c,l,\leq)$, where (N,S,c) is a (multiple, hyper-)graph, and $(S,l,\leq)$ is a history.

The replacement operation can be extended to deal with the spatial aspects by using a construction analogous to one developed in the algebraic theory of graph grammars /Ehri83/. More precisely, in the algebraic approach a production

$$p=(B1 \xleftarrow{b1} K \xrightarrow{b2} B2)$$

consists of three graphs B1, K and B2 and two graph homomorphisms b1 and b2. Graphs B1 and B2 represent the usual left and right members (B1 will be replaced by B2), while the "interface" graph K and the two homomorphims b1 and b2 are used for defining the "embedding", i.e. the connections of B2 with the rest of the graph. A distributed history h is analogous to a production P if we identify B1 and B2 with the heads and tails of h. Furthermore, if we assume that K consists only of ports and that b1 and b2 are injective, the role of K, b1 and b2 (i.e. specifying corresponding ports) can be played by those ports of the history connected to both heads and tails. These ports are called external ports.

In the algebraic theory, given two productions p' and p", a new production

$$p=p'*_{R}p''$$

can be constructed, whose application is equivalent to the successive applications of p' and p" /Ehri83/. The graph R specifies which parts of B2' and B1" will be identified.

The replacement operation between distributed histories can be defined in exactly the same way. The events and the partial ordering relation of the result are of course obtained as for non distributed histories in Level 1..

Atomic histories are also defined as in the non distributed case, but with one additional constraint. In fact we require as before the existence of one event greater than all heads and smaller then all tails. Furthermore, this event must be connected to all and only ports which are not external.

Non external ports of atomic distributed histories are called synchronization ports. The intended meaning of the above constraint is that all ports which are deleted and created by the transition are clearly affected by the event. Viceversa, a synchronization on some port, being a point-like, completely centralized transition, must involve all processes connected to that port. Thus external ports, which represent the boundary with the external world, cannot be loci of synchronization.

In Fig. 5 we see a set Z of three distributed, atomic histories and a three-step computation. Notice that ports are represented as variables and atomic histories are written as usual in linear form. Note also that in the distributed histories in Fig. 5b) and c) port y_1 is external, while in d) there are no external ports.

Also for distributed histories the problem arises of generating atomic histories through a synchronization mechanism. This is possible using algebraic constructions like the gluing star /Corr83, CDM84, BFH84/.

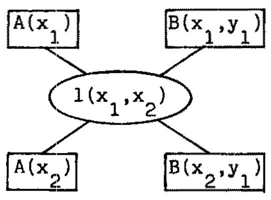
A formalism called GDS, essentially consistent with the one outlined here, is described in detail in /CaMo83, DeMo83, DeMo84b/.

$A(x),B(x,y) \xrightarrow{l(x,x')} A(x'),B(x',y)$

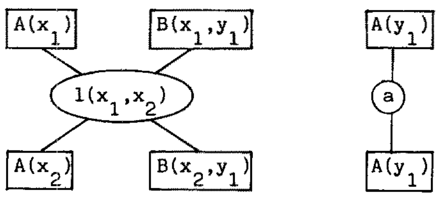
$B(x,y),A(y) \xrightarrow{r(y,y')} B(x,y'),A(y')$

$A(z) \xrightarrow{a} A(z)$

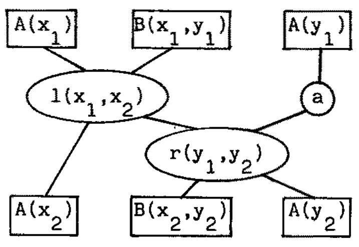
a)



b)



c)



d)

Fig. 5. A set Z of atomic histories (a)) and a three-step computation (b), c), d)).

3.6. Abstract data types and logic programming

In this section, we refine the model of Level 1.2. of Section 3.3. in a different direction. So far the alphabets E of process types and A of observable actions were unstructured, yet infinite. Thus no explicit notion of data type was available. In the present Level 1.2.2. we parametrize the symbols of both alphabets with (tuples of) terms of an Herbrand universe. Logic programming axiomatizations of data in terms of Horn clauses can be given by using an enriched version of the productions of Section 3.3., thus providing a well-known tool for defining abstract data types. For a comprehensive reference on logic and logic programming see /Robi79/.

Let E and A be disjoint sets of positive literals, i.e. standard first-order predicates on terms with variables. A history with variables is meant to represent the set of all ground histories obtained by instantiating the variables with ground terms, i.e. terms without variables. The operation of replacement between two histories

h_1 <u>before</u> h_2 <u>gives</u> h

permits now the (most general) instantiation of variables of both h_1 and h_2 before matching the subsets of the heads of h_2 and of the tails of h_1. This is the standard operation of unification, applied to tuples of predicates. Of course the resulting history represents exactly the set of ground histories derivable by the old replacement operation.

Let us comment on the atomic histories with variables. An atomic history with a single head and no event is exactly a Horn clause (with a reversed arrow!). Atomic histories with many heads and no events are strictly related to the model of so-called Generalized Horn Clauses already studied in the literature /DeDi83, FLP84/. Thus our computations without events can be interpreted as proofs in a logic framework. Note however that in our case new hypotheses can be added during the proof as needed, due to the partiality of our replacement operation (as defined in Section 3.1.).

The mechanism (defined at Level 2.2.) of combining productions (decorated with protocols) according to a synchronization function, to obtain atomic histories can be generalized to histories with variables. Also protocols are literals (of course possibly sharing variables with heads and tails) while the synchronization function is defined on predicate symbols only. Therefore a single unification problem can be set up for both synchronization and history replacement.

A similar mechanism has been defined in the literature by Monteiro /Mont84/. It uses essentially the CCS communication mechanism and generates histories without events.

As a general comment, in our approach we emphasize the description of the temporal evolution of concurrent distributed systems (as a partial ordering of events) because we aim at having a process-oriented semantics rather than an input-output semantics. Along this line we can rely also on our results on semantics and properties of infinite computations /DeMo84a, b/.

In Fig. 6 we see an example of a simple system consisting of a producer P and a consumer C connected by a queue Q. The producer generates a random natural number by iterating a successor operation s. When communication with the queue takes place, the current number is enqueued and the counter is reset. The queue can output its front value by communicating with the consumer, provided the latter is in a ready state. Eventually the consumer prints the value. The queue is implemented as a pair of stacks. When the output stack is empty, it receives the reversed content of the input stack. Note that the atomic histories handling the stacks do not produce events, and thus the observable behaviour of Q is only that of a queue. It may be interesting to examine the results of the infinite computations in this example (with heads P,Q and C) when different metrics are chosen to define the limits. It is easy to see that all computations are vital and thus have a result in out first metric space. When local fairness is required (namely our third metric is used) the producer cannot count forever, all the computations are also partial deadlock free and thus the limits have no tails. Note that in this case producer P realizes a choice operation, i.e. generates an unbound natural number still guaranteeing termination.

We conclude this section by showing the whole of our level structure:

Level 1. Concurrent histories

    Level 1.1. Petri nets

    Level 1.2. Productions - synchronization

        Level 1.2.1. Spatial structure - GDS

        Level 1.2.2. Abstract data types - logic programming

    Level 1.3. Inference rules of CCS.

$$P(n) \xrightarrow{count} P(s(n))$$

i) $$P(n) \xrightarrow{read1(n)} P(0)$$

ii) $$Q(S_1,S_2) \xrightarrow{read2(n)} Q(cons(n,S_1),S_2)$$

$$Q(cons(n,S_1),nil) \longrightarrow R(cons(n,S_1),nil)$$

$$R(cons(n,S_1),S_2) \longrightarrow R(S_1,cons(n,S_2))$$

$$R(nil,S_2) \longrightarrow Q(nil,S_2)$$

iii) $$Q(S_1,cons(n,S_2)) \xrightarrow{write1(n)} Q(S_1,S_2)$$

iv) $$C(ready) \xrightarrow{write2(n)} C(full(n))$$

$$C(full(n)) \xrightarrow{print(n)} C(ready)$$

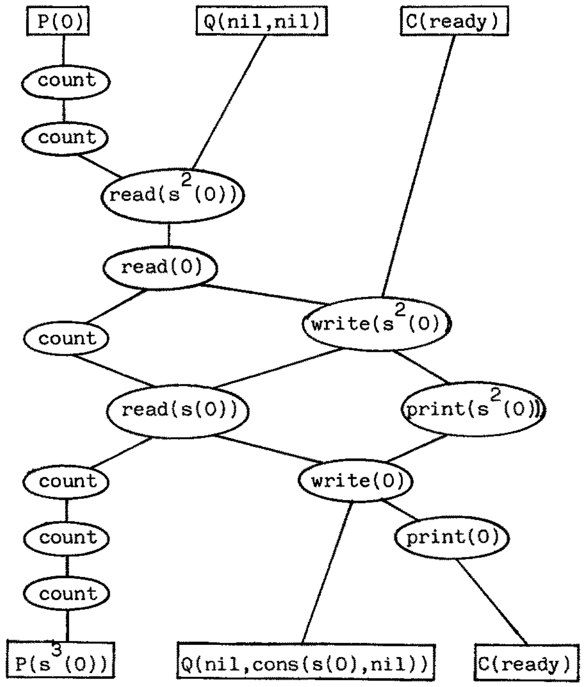
f(read1,read2)=read

f(write1,write2)=write

i)+ii) $$P(n),Q(S_1,S_2) \xrightarrow{read(n)} P(0),Q(cons(n,S_1),S_2)$$

iii)+iv) $$Q(S_1,cons(n,S_2)),C(ready) \xrightarrow{write(n)} Q(S_1,S_2),C(full(n))$$

a)



b)

Fig.6. Productions, synchronization function and atomic histories (a)), and a derived history (b)) for a producer-consumer example.

Acknowledgements

The authors wish to thank Sonia Raglianti and Silvia Giannini for having accurately typed this manuscript in an incredibly short time.

References

/BCG83/ Balzer,R., Cheatham,T.E. and Green,C., Software Technology in the 1990's: Using a New Paradigm, Computer 16, 11 (1983), 39-45.

/BFH84/ Boehm,P., Fonio,H., and Habel,A. Amalgamation of Graph Transformations with Applications to Synchronization, to appear in Proc. CAAP'85, Berlin, March 25-29, 1985, Springer-Verlag, Berlin.

/BHR84/ Brookes,S.D., Hoare,C.A.R., and Roscoe,A.W. A Theory of Communicating Sequential Processes, J. ACM 31, 3 (1984), 560-599.

/BuLa84/ Burstall,R., and Lampson,B. A Kernel Language for Abstract Data Types and Modules, Proc. Symp. on Semantics of Data Types (G. Kahn, D.B. MacQueen, and G. Plotkin Eds), Sophia Antipolis (France), June 1984, LNCS, 173, Berlin, 1984, pp. 1-50.

/Bram83/ Brams, G.W. Réseaux de Petri: Théorie et Pratique, Vol. I and II, Masson, Paris, 1983.

/CaMo83/ Castellani,I., and Montanari,U. Graph Grammars for Distributed Systems, Proc. 2nd Int. Workshop on Graph-Grammars and their Applications to Computer Science (H. Erigh, M.A. Nagel, and G. Rozenberg Eds), LNCS, 153, Springer-Verlag, Berlin 1983, pp. 20-83.

/CDM84/ Corradini,A., Degano,P., and Montanari,U. Specifying Highly Concurrent Data Structure Manipulation, to appear in Proc. ACM Int. Symp. on Computing, Firenze, March 1985, North-Holland.

/CFM82/ Castellani,I., Franceschi,P., and Montanari,U. Labeled Event Structures: A Model for Observable Concurrency, Proc. IFIP TC 2 - Working Conference: Formal Description of Programming Concepts II (D. Bjørner Ed), Garmisch - Partenkirchen. 1982. North-Holland. Amsterdam, 1983, pp. 383-400.

/Corr84/ Corradini,A. Aspetti Modellistici ed Implementativi di GDS, un Formalismo per la Specifica di Sistemi Distribuiti, Master Thesis, Computer Science Dept., Univ. of Pisa, Pisa, April 1984.

/DDM84/ Degano,P., De Nicola,R., and Montanari,U. Partial Ordering Derivations for CCS, submitted for publication.

/DeDi83/ Degano,P., and Diomedi,S. A First Order Semantics of a Connective Suitable to Express Concurrency, Proc. 2nd Logic Programming Workshop (L.M. Pereira Ed.), Albufeira (Portugal), 1983, pp. 506-517.

/DeMo83/ Degano,P., and Montanari,U. A Model for Distributed Systems Based on Graph Rewriting, Note Cnet 111, Computer Science Dept., Univ. of Pisa, Pisa, 1983.

/DeMo84a/ Degano,P., and Montanari,U. Liveness Properties as Convergence in Metric Spaces, Proc. 16th Annual ACM SIGACT Symposium on Theory of Computing, April 30 - May 2, 1984, Washington, DC, pp. 31-38.

/DeMo84b/ Degano,P., and Montanari,U. Distributed Systems, Partial Orderings of Events, and Event Structures, Lecture Notes of the 1984 International Summer School on Control Flow and Data Flow - Concepts of Distributed Programming (M. Broy Ed.), LNCS, Springer-Verlag, Berlin, 1984.

/Ehri83/ Ehrig,H. Aspects of Concurrency in Graph Grammars, Proc. 2nd Int. Workshop on Graph-Grammars and their Applications to Computer Science (H. Ehrig, M.A. Nagel, and G. Rozemberg Eds), LNCS, 153, Springer-Verlag, Berlin, 1983.

/FLP84/ Falaschi,M., Levi,G., and Palamidessi,C. A Synchronization Logic: Axiomatics and Formal Semantics of Generalized Horn Clauses, Info. and Co. 60 (1984), 36-69.

/GRAS84/ ESPRIT Pilot Project 125 - GRASPIN, Foundation Study for a Personal Software Development Workstation Prototype, Pilot Phase Project Report, October 1984.

/JaVa80/ Jantzen,M., and Valk,R. Formal Properties of Place/Transition Nets, In: Net Theory and Applications (W. Brauer Ed.), LNCS, 84, Springer-Verlag, Berlin 1979, pp. 165-212.

/LTS79/ Lauer,P.E., Torrigiani,P., Shields,M.W. COSY: A Specification Language based on Path Expressions, Acta Informatica, 12 (1979), 109-158.

/Miln80/ Milner,R. A Calculus of Communicating Systems, LNCS, 92, Springer-Verlag, Berlin 1980.

/Miln83/ Milner,R. Calculi for Synchrony and Asynchrony, TCS 25 (1983), 267-310.

/Mont84/ Monteiro,L. A Proposal for Distributed Programming in Logic, In: Implementations of PROLOG (J.A. Campbell Ed.), Ellis Horwood, Chichester, 1984, pp. 329-340.

/Plot83/ Plotkin,G.D. An Operational Semantics for CSP, Proc. IFIP TC 2-Working Conference: Formal Description of Programming Concepts II (D. Biørner Ed.), Garmisch-Partenkirchen, June 1982, North-Holland, Amsterdam 1983, pp. 199-223.

/Robi79/ Robinson,J.A. Logic: Form and Function, Edinburgh University Press, 1979.

/Roev84/ de Roever, W.-P. The Quest for Compositionality - A Survey of Assertion--Based Proof Systems for Concurrent Programs, to appear in Proc. IFIP Working Conference on the Role of Abstract Models in Information Processing, Vienna, January 30 - February 1, 1985.

/Spec79/ Abstract Software Specifications, 1979 Copenhagen Winter School Proc. (D. Bjørner Ed.), LNCS, 86, Springer-Verlag, Berlin, 1980.

/Spec83/ STL/SERC Workshop on Analysis of Concurrent Systems, September 12-16, 1983, Cambridge (U.K.).

/Spec84/ Combining Specification Methods, Proc. 1984 Workshop on Formal Software Development, May 20-26, 1984, Nyborg (DK).

/Wins82/ Winskel,G. Event Structure Semantics for CCS and Related Languages, Proc. 9th ICALP (M. Nielsen and E.M. Schmidt Eds), LNCS, 140, Springer-Verlag, Berlin 1982, pp. 561-576.