Math. Struct. in Comp. Science (1993), *vol.* 11, *pp.* 1–000 Copyright © Cambridge University Press

The Dynamics of Wait-Free Distributed Computations

Eric Goubault

C.N.R.S & Ecole Normale Supérieure
45 rue d'Ulm
75230 PARIS Cedex 05, FRANCE

Received 15 December 1996

1. Introduction

The work reported here is concerned with the *robust* or *fault-tolerant* implementation of distributed programs. More precisely, we are interested in *wait-free* implementations on a distributed machine composed of several units communicating through a shared memory via atomic read/write registers (described in Section 2). This means, in the case of two such units, that the processes executed on the two processors (say P and P') must be as loosely coupled as possible so that even if one fails to terminate, the other will carry on computation and find a correct partial result. This excludes all mutual exclusion constructs such as semaphores, monitors etc. Wait-freeness is also intended to help solve an efficiency problem: if one of the processors is much slower than the other, can we still implement a given function in such a way that the fast process will not have to wait too much for the slow one?

This field of distributed computing has received up to now considerable attention. Typically, one is interested in implementing a distributed database in which remote transactions do not have to wait for each others. The kind of functions we have to consider then is more like coherence relations between the possible local inputs on each processor and the final global output of the machine. For instance, when two transactions wish to change the same shared item in the database in an asynchronous manner, one has to choose which transaction will get the leading rôle, to keep the database coherent. This is the well known *consensus problem*. Formally, if we represent the values of the shared items by integers then the consensus problem is the input/output relation $\Delta \subseteq (\mathbb{Z} \times \mathbb{Z}) \times (\mathbb{Z} \times \mathbb{Z})$ defined as follows, given that a pair of integers represents a pair of local values on P, P'.

For all integers i, $(i,i)\Delta(i,i)$ (a). This means that if P and P' start with the same local input value i, then they must end with the same output value i as well. This corresponds to the fact that they can only agree on the value i in that case.

For all i, j, $(i,j)\Delta(i,i)$ (b) : if P and P' start with different local input values, say i, j, then P and P' can agree on value i.

For all i, j, $(i,j)\Delta(j,j)$ (c) : P and P' can also agree on value j.

What if now one of the two processors fails to terminate? If we represent failure by the symbol $\perp$, then the coherence relation Δ has to be extended so that it expresses the behaviour of the system in nasty cases.

For all i, $(i,\perp)\Delta(i,\perp)$ (d): if P' fails then P must terminate and stick to its local value i.

We should also assume for all j, $(\perp,j)\Delta(\perp,j)$ (e) : if P fails then P' must terminate and stick to its local value j.

In fact, it is well known that this relation cannot be implemented in a wait-free manner on a shared memory machine with atomic read/write registers (FLP85), whereas the following approximate consensus, called binary pseudo-consensus in (Her94), has a solution:

(a')For all i, j booleans, $(i,j)\Delta(i,i)$, $(i,j)\Delta(j,j)$. This is the same as (a), (b) and (c) (for boolean values 0 and 1).

(b')$(0,1)\Delta(1,0)$.

(c')Same as (d) and (e).

We have just slightly relaxed the agreement problem by adding rule (b') specifying that we could agree except for input $(0,1)$ where a minor error is tolerated (and we have also restricted to the subdomain $\{0,1\} \subseteq \mathbb{Z}$). We can implement this one in a wait-free manner, as will be shown in Section 11.1.1.

We follow here the geometric view on distributed computation used in recent litterature in distributed protocols (BG93; Cha90; Her94; HR94; HR95; HS93; HS94; SZ93) and in some ways in recent litterature in semantics of concurrency (Gou95; Gou96; Goult; Pra91; vG91; Gun94). The idea is that wait-free relations exhibit some geometrical properties (Section 9). We give another way of proving this (with respect to the way of M. Herlihy, N. Shavit and S. Rajsbaum), starting with a semantics of a shared memory language, bringing these considerations close to the semantics and language people. We actually are first interested in the case where we have two communicating units. This case is the simplest possible and we will be able to discuss in a precise manner all the behaviour of these machines.

We begin by defining a precise language accessing shared memory though atomic reads and writes. After giving a precise definition of wait-freeness, we prove that all programs written in that language are wait-free. In the course of proving this, we will need to separate out the part of the semantics which deals with control flow and the one which deals with the information flow (the values of the variables). Wait-freeness is a property on the control flow basically asserting that it should be invariant under all possible permutations of processors, at each stage of the execution, hence if one action fails, there is always a way to escape from failure by rescheduling actions yet to be executed, to be the ones of the non-faulty processors. This will be explained geometrically as being semantics which are essentially contractible spaces.

But the topology of the whole semantics can be studied explaining the link between the possible schedules and the final states of the memory. Traversing the set of schedules we have to go through critical points (determining the interactions between the processes). The fact that these interactions made of reads and writes can only change a small amount of the memory at each time, implies that there is a topological invariant throughout the

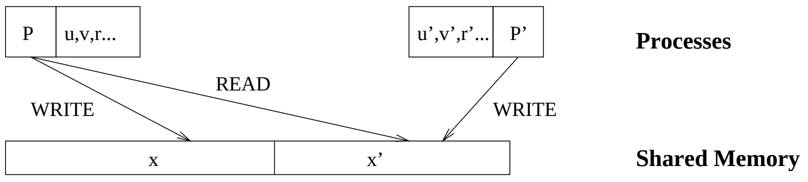

Fig. 1. Sketch of a shared memory machine with atomic read/write registers.

dynamics, namely, some form of connectedness is preserved along the executions of a program.

This has actually a reciprocal. Once again we can treat this exhaustively in the case of two processors. Basically, an input/output relation that preserves this kind of topological invariant we determined beforehand, is the "denotation" of a program in our small read/write language. This should be seen as some kind of full-abstraction result, and also a precise statement of the expressive power of wait-free atomic read/write shared memory distributed architectures.

We derive a different algorithm than the one of (Her94; HS94) based on the participating set algorithm of (Bor95) directly from the semantics of our language (Section 11.3). Its short proof stems directly from its construction. Then, after giving a few examples, we compare both algorithms (Section 11.4) and show that ours gives the programs with the minimum number of comparisons and accesses to the shared memory for all possible executions, hence produces the most efficient code for computing any wait-free binary relation.

We then give a general methodology for dealing with these kinds of results from the operational semantics to the geometric invariants of computations we might expect.

We sketch a possible generalisation of this to the n-processor case. The geometric and semantic phenomena are not different from the 2-processor case but the combinatorics is much more intricate.

Finally, we treat a general computability result concerning atomic read/write shared memory plus a test&set primitive. It can be shown now that any "finite" binary relation can be computed, and a general algorithm for doing so is sketched in Section 14.

2. A Simple Asynchronous Distributed System

We consider a shared memory machine with two processors such as the one pictured in Figure 1. The shared memory is formalized by a collection of registers $V = \{x, y\}$. Processor P (resp. P') has a local memory composed of locations $u, v, r \cdots$ (resp. $u', v', r' \cdots$). All reads and writes are done in an asynchronous manner on the shared memory. There is no conflict in reads, nor in writes since we ensure that the writes of distinct processors are made on distinct parts of the shared memory (P is only allowed to write on x, P' is only allowed to write on x').

3. Syntax

We use the following syntax for the shared memory language handling this machine. We first have a grammar for instructions I, and then another one for processes P,

$$\begin{array}{rcl} I & := & \mathit{update} \\ & | & \mathit{scan} \\ & | & r = f(r_1, \cdots, r_n) \end{array}$$

where c is a local register or a value (in $\mathbb{Z}$), $r, r_1, \cdots, r_n$ are local registers and f is any partial recursive function.

$$\begin{array}{rclll} P & := & I & & \\ & | & \mathit{case} & (u_1, u_2, \ldots, u_k) & \mathit{of} \\ & & & (a_1^1, a_2^1, \ldots, a_k^1): & P \\ & & & \ldots & \\ & & & (a_1^n, a_2^n, \ldots, a_k^n): & P \\ & & & \mathit{default}: & P \\ & | & P; P & & \end{array}$$

where r is any local register. We also suppose that

$$(a_1^i, \cdots, a_k^i) = (a_1^j, \cdots, a_k^j) \Rightarrow i = j$$

Programs are $Prog := (P \mid P)$ (we are considering programs on two processors only).

update is the instruction that writes the local value u (resp. v') of processor P (resp. P') in the shared variable x (resp. x').

scan reads the shared array in one round and stores it into a local register of the process in which it is executed. *scan* executed in P (resp. P') stores x' (resp. x) in v (resp. u').

$r = f(r_1, \cdots, r_n)$ computes the partial recursive function f

case is the ordinary case statement on any tuple of local registers, with any finite number of branches allowed.

; is the sequential composition of processes.

| is the parallel composition of processes.

4. A first Concrete Semantics

We denote both the shared and local stores by ρ which is a function from $V \cup (\cup_i V_i)$ to $\mathbb{Z}$, the domain of values. The semantics is given in terms of a transition system generated by the rules below. The states of the transition system are pairs $(\{P, P'\}, \rho)$ where P (respectively P') is the text of the program yet to be executed on the first processor (respectively second processor) and ρ is the value of the global and local memories at this point of the computation.

(update)

$$(\{\mathit{update}; R, P'\}, \rho) \xrightarrow{\mathit{update}_P} (\{R, P'\}, \rho[x \leftarrow u])$$

$(scan)$

$$(\{scan; R, P'\}, \rho) \xrightarrow{scan_P} (\{R, P'\}, \rho[v \leftarrow x'])$$

$(calc)$

$$(\{(r = f(r_1 \cdots r_n)); R, P'\}, \rho) \xrightarrow{calc_P} (\{R, P'\}, \rho[r \leftarrow f(r_1 \ldots r_n)])$$

$(case)$

If $\exists k, \forall i, \rho(u_i) = a_i^k$,

$$\left(\left\{\left(\begin{array}{l} case\ (u_1 \ldots u_k)\ of \\ (a_1^1 \ldots a_k^1) :\ P_1 \\ \ldots \\ (a_1^n \ldots a_k^n) :\ P_n \\ default :\ P \end{array}\right); R, P'\right\}, \rho\right) \xrightarrow{case_P} (\{P_k; R, P'\}, \rho)$$

Otherwise,

$$\left(\left\{\left(\begin{array}{l} case\ (u_1 \ldots u_k)\ of \\ (a_1^1 \ldots a_k^1) :\ P_1 \\ \ldots \\ (a_1^n \ldots a_k^n) :\ P_n \\ default :\ P \end{array}\right); R, P'\right\}, \rho\right) \xrightarrow{case_P} (\{P; R, P'\}, \rho)$$

We also add the obvious symmetric rules where we interchange the rôles of P and P'. The respective actions are denoted with a P' subscript, when needed. We call solo execution of P (respectively P') all paths composed of actions of the form a_P, $a = scan, update, calc, case$ (respectively $a_{P'}$). A maximal path is a path for which there is no path containing it strictly.

5. A first Abstraction

The basic idea here is to look only at the input/output relations that a given program induces. This is the ordinary denotational view on the semantics of the program (CC92). Formally we define an abstract domain of denotations of programs as

$$\mathcal{D} = (\{P\} \times \mathbb{Z}_\perp)^2 \cup (\{P'\} \times \mathbb{Z}_\perp)^2$$

Note the slight difference with a standard (relational) denotational semantics. Instead of computing relations between pairs of values computed by P and Q, we look at solo executions of P and solo executions of Q, i.e. we compute relations between values of P or relations between values of Q. Let us be more formal now.

Let

$$\begin{array}{rcl} p_I(\{R, S\}, \rho) & = & (P, \rho(u)) \\ p_O(\{R, S\}, \rho) & = & (P, \rho(x)) \\ q_I(\{R, S\}, \rho) & = & (P', \rho(v)) \\ q_O(\{R, S\}, \rho) & = & (P', \rho(y)) \end{array}$$

Then the abstraction α from the transition systems defining the semantics of our language (forming a lattice $\mathcal{T}$ with the inclusion of transition systems as order) to the domain $\mathcal{D}$ (which is a lattice with the inclusion of relations as order) is such that

$$\begin{aligned} \alpha : \; T \longrightarrow \alpha(T) = & \bigcup \{(p_I(s), p_O(s'))/s \rightarrow^* s' \text{ is a maximal solo execution of } P \text{ in } T\} \\ & \cup \bigcup \{(q_I(s), q_O(s'))/s \rightarrow^* s' \text{ is a maximal solo execution of } Q \text{ in } T\} \end{aligned}$$

In fact, we will need two other abstractions of the concrete semantics. The first one is the abstraction of the semantics to the control flow (α_c) and the second one to the information flow (α_i). The interplay between these two will enable us to fully characterize the denotational abstraction α.

α_c of a transition system generated by the SOS rules of the concrete semantics only retains the control part of the states, i.e. is a folding of the transition system on states, such that $\alpha_c(\{P, P'\}, \rho) = \{P, P'\}$.

The abstract semantics of a term $\{P, P'\}$ executed from environment ρ is denoted $[\![P, P']\!]^\rho_c$. We have the following rules for the abstract semantics,

(update)

$$[\![update; R, P']\!]^\rho_c = \{update; R, P'\} \xrightarrow{update_P} \{R, P'\} \cup [\![R, P']\!]^{\rho[x \leftarrow u]}_c$$

(scan)

$$[\![scan; R, P']\!]^\rho_c = \{scan; R, P'\} \xrightarrow{scan_P} \{R, P'\} \cup [\![R, P']\!]^{\rho[v \leftarrow x']}_c$$

(calc)

$$[\![(r = f(r_1 \cdots r_n)); R, P']\!]^\rho_c = \{(r = f(r_1 \cdots r_n)); R, P'\} \xrightarrow{calc_P} \{R, P'\} \cup [\![R, P']\!]^{\rho[r \leftarrow f(r_1 \ldots r_n)]}_c$$

(case)

If $\exists k, \forall i, \rho(u_i) = a^k_i$,

$$[\![\begin{pmatrix} case\ (u_1 \ldots u_k)\ of \\ (a^1_1 \ldots a^1_k) :\ P_1 \\ \cdots \\ (a^n_1 \ldots a^n_k) :\ P_n \\ default :\ P \end{pmatrix}; R, P']\!]^\rho_c \xrightarrow{case_P} \{P_k; R, P'\} \cup [\![P_k; R, P']\!]^\rho_c$$

Otherwise,

$$[\![\begin{pmatrix} case\ (u_1 \ldots u_k)\ of \\ (a^1_1 \ldots a^1_k) :\ P_1 \\ \cdots \\ (a^n_1 \ldots a^n_k) :\ P_n \\ default :\ P \end{pmatrix}; R, P']\!]^\rho_c \xrightarrow{case_P} \{P; R, P'\} \cup [\![P; R, P']\!]^\rho_c$$

α_i of a transition system generated by the SOS rules of the concrete semantics only retains the information part of the states, i.e. $\alpha_i(\{P, P'\}, \rho) = \rho$.

This abstract semantics is given by the following abstract rules.

(update)

$$(\rho) \xrightarrow{update_P} (\rho[x \leftarrow u])$$

$(scan)$

$$(\rho) \xrightarrow{scan_P} (\rho[v \leftarrow x'])$$

$(calc)$

$$(\rho) \xrightarrow{calc_P} (\rho[r \leftarrow f(r_1 \dots r_n)])\text{for all computable } f$$

$(case)$

$$(\rho) \xrightarrow{case_P} (\rho)$$

These are sound abstractions of the concrete semantics.

6. A second Concrete Semantics: HDA

In fact, we do need to know more about how asynchronous the execution can be. Intuitively, our machine is designed to be entirely asynchronous: there is no locks on any of the locations in the shared memory that would make one processor wait for the other. All programs written on this machine are wait-free in that sense (Lyn96) since we cannot emulate active polling because there is no loop construct. This statement should be somehow reflected in the semantics of the language, an aspect which is missing in the interleaving semantics of Section 4, if we want to reason formally about this asynchronous machine.

In order to do this, we use HDA (Pra91; vG91; GJ92; Gou93; Goult) to model the language. Basically we add up 2-transitions (transitions of dimension 2) to the transitions of dimension 1 already specified by the operational semantics of Section 4 that indicate that the 1-transitions at its boundaries are executed in a concurrent manner. Formally, 2-transitions have two starting 1-transitions (respectively two ending 1-transitions) whereas 1-transitions have one starting 0-transition, or state (respectively one ending 0-transition, or state). Hence we specify a 2-transition A from the 1-transitions a, b to the 1-transitions a', b' by the notation:

$$a, b \xrightarrow{A} a', b'$$

In fact this notation is a direct abstraction of the general definition of HDA as found in e.g. (Goult), that we recall and explain below.

Definition 1. An unlabeled semi-regular HDA is a collection of sets M_n ($n \in \mathbb{N}$) together with functions

$$\begin{tikzcd} M_{p,q} \arrow[r, "d_i^0"] \arrow[d, "d_i^1"'] & M_{p-1,q} \\ M_{p,q-1} & \end{tikzcd}$$

for all $n \in \mathbb{N}$ and $0 \leq i, j \leq n-1$, such that

$$d_i^k \circ d_j^l \quad = \quad d_{j-1}^l \circ d_i^k$$

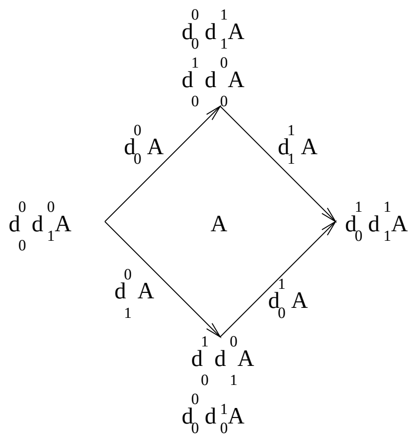

Fig. 2. The 2-transition A and its boundaries

$(i < j \; and \; k, l = 0, 1)$ and $\forall n, m \; n \neq m, \quad M_n \cap M_m = \emptyset$.

Elements x of $M_{p,q}$ ($\dim x = p + q = n$) are called n-transitions (or states if $n = 0$). d^0_i (respectively d^1_i) are called the start boundary operators (respectively the end boundary operators). For a transition such as the 2-transition $A \in M_{1,1}$ (for instance) denoted as,

$$a, b \xrightarrow{A} a', b'$$

the start 1-transitions are $a \in M_{0,1}$ and $b \in M_{0,1}$, for instance,

$$\begin{array}{rcl} d^0_0(A) & = & a \\ d^0_1(A) & = & b \end{array}$$

and the end 1-transitions are $a' \in M_{1,0}$ and $b' \in M_{1,0}$ with,

$$\begin{array}{rcl} d^1_0(A) & = & a' \\ d^1_1(A) & = & b' \end{array}$$

The 1-transitions a, b, a' and b' have also start and end 0-transitions or states, that we can write using the format of last definition as,

$$\begin{array}{rcl} d^0_0(a) & = & \alpha \in M_{-1,1} \\ d^0_0(b) & = & \alpha \\ d^0_0(a') & = & \gamma \in M_{0,0} \\ d^0_0(b') & = & \beta \in M_{0,0} \\ d^1_0(a) & = & \beta \\ d^1_0(b) & = & \gamma \\ d^1_0(a') & = & \delta \in M_{1,-1} \\ d^1_0(b') & = & \delta \end{array}$$

One can check on this example that the commutation rule defining semi-regular HDA is indeed verified (as shown on Figure 2) on that particular example, for instance,

$$\begin{array}{rcl} d_0^1(d_1^0(A)) & = & d_0^1(b) \\ & = & \gamma \\ & = & d_0^0(a') \\ & = & d_0^0(d_0^1(A)) \end{array}$$

Let us review the rules for 2-transitions now. First, we need a little lemma on the interleaving semantics.

Lemma 1. Let

$$\begin{array}{rcl} a & = & (\{t;P',s;Q'\},\rho) \xrightarrow{t} (\{P',s;Q'\},\rho_1) \\ b & = & (\{t;P',s;Q'\},\rho) \xrightarrow{s} (\{t;P',Q'\},\rho_2) \\ a' & = & (\{t;P',Q'\},\rho_2) \xrightarrow{t} (\{P',Q'\},\rho_3) \\ b' & = & (\{P',s;Q'\},\rho_1) \xrightarrow{s} (\{t;P',Q'\},\rho_4) \end{array}$$

with $(t,s) \neq (update_P, scan_{P'})$ and $(t,s) \neq (scan_P, update_{P'})$. Then $\rho_3 = \rho_4$.

Proof. By simple case analysis, using the semantics of Section 4. □

Basically, this lemma states that all actions but *update* and *scan* commute with each other, making them possible candidates for being run in a truly concurrent manner (hence delimiting 2-transitions). For an *update* and a *scan* run in different orders, we might get different environments at the end. Nevertheless, the rule (*interference*) will state that these two actions are run in a truly concurrent manner (no synchronisation between them is required by the machine), whereas some other systems (like those which use semaphores to handle read/write conflicts) would be specified as sequential histories only (no 2-transition indicating asynchrony).

(*no interference*)

For 1-transitions a, b, a' and b' as in Lemma 1, or more generally, such that $\rho_3 = \rho_4$,

$$a, b \xrightarrow{t \otimes s} a', b'$$

(*interference*)

For 1-transitions a, b, a', b' of the form,

$$\begin{array}{rcl} a & = & (\{t;P',s;Q'\},\rho) \xrightarrow{t} (\{P',s;Q'\},\rho_1) \\ b & = & (\{t;P',s;Q'\},\rho) \xrightarrow{s} (\{t;P',Q'\},\rho_2) \\ a' & = & (\{t;P',Q'\},\rho_2) \xrightarrow{t} (\{P',Q'\},\rho_3) \\ b' & = & (\{P',s;Q'\},\rho_1) \xrightarrow{s} (\{P',Q'\},\rho_4) \end{array}$$

with $t = update_P$ and $s = scan_{P'}$ and such that $\rho_3 \neq \rho_4$ introduce new 1-transitions,

$$\begin{array}{rcl} \epsilon_1 & = & (\{t;P',Q'\},\rho_2) \xrightarrow{\epsilon} (\{P',Q'\},\rho_4) \\ \epsilon_2 & = & (\{P',s;Q'\},\rho_1) \xrightarrow{\epsilon} (\{P',Q'\},\rho_3) \end{array}$$

then,

(1)

$$a, b \xrightarrow{t \otimes s} \epsilon_1, b'$$

(2)

$$a, b \xrightarrow{s \otimes t} a', \epsilon_2$$

The 2-transition created in (1) is abstracting the asynchronous behaviours of t and s in which s terminates after t. The one created in (2) represents the asynchronous executions in which t terminates after s.

7. A second Abstract Semantics: "Cut" semantics

First, we are able to generalise α_c and α_i to deal with 2-transitions.

Basically, in the case where t and s are non-conflicting, taking the notations of Lemma 1,

$$[\![t; P', s; Q']\!]_c^\rho = \ a, b \xrightarrow{t \otimes s} a', b' \cup [\![P', s; Q']\!]_c^{\rho_1} \cup [\![t; P', Q']\!]_c^{\rho_2}$$

In the case where t and s are conflicting, we are generating a unique 2-transition $t \otimes s$, but disconnect the executions "above" and "below", since the two histories are different in an essential manner,

$$[\![t; P', s; Q']\!]_c^\rho = \ a, b \xrightarrow{t \otimes s} a', b' \cup [\![P', s; Q']\!]_c^{\rho_1} \cup ([\![t; P', Q']\!]_c^{\rho_2})'$$

where the only intersection between $[\![P', s; Q']\!]_c^{\rho_1}$ and $([\![t; P', Q']\!]_c^{\rho_2})'$ is the point $\{P', Q'\}$[†].

This is still a sound abstraction of the concrete semantics.

As for α_i we have to complete the abstract semantics as follows,

If t and s are two non-conflicting 1-transitions, we have the following 2-transition,

$$a^i, b^i \xrightarrow{t \otimes s} a'^i, b'^i$$

where a^i, b^i, a'^i, b'^i are the respective transitions of the information flow semantics.

And if t and s are conflicting 1-transitions, we have two new 1-transitions,

$$\begin{array}{rcl} \epsilon_1^i & = & (\rho_2) \xrightarrow{\epsilon} (\rho_4) \\ \epsilon_2^i & = & (\rho_1) \xrightarrow{\epsilon} (\rho_3) \end{array}$$

then, we have the two following 2-transitions,

$$a^i, b^i \xrightarrow{t \otimes s} \epsilon_1^i, b'^i$$

$$a^i, b^i \xrightarrow{s \otimes t} a'^i, \epsilon_2^i$$

Finally the α abstraction can be completed as well. Now we can relate not only inputs with outputs of solo executions but also pairs of inputs to pairs of outputs for executions of both P and P'. Formally, still using the p_I, p_O, q_I and q_O projection maps of Section 5, we define α' to be the following abstraction of the HDA semantics, defined to be in value in the domain $\mathcal{D}' = (\mathbb{Z}_\perp \times \mathbb{Z}_\perp)^2$. Let us call $p_1(s, s')$ any 1-path such that the first

† This "trick" will be explained in Section 12.

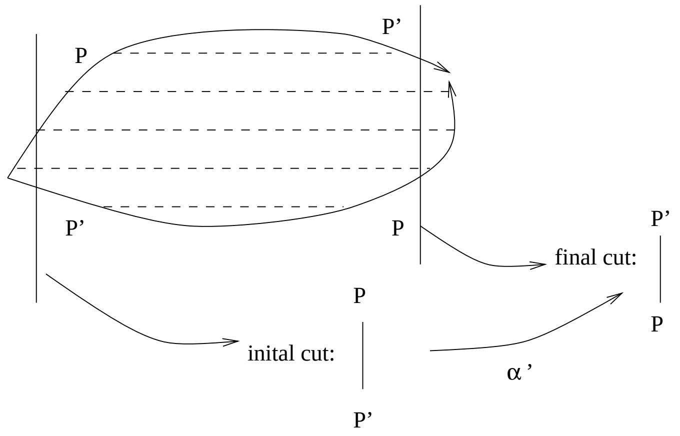

Fig. 3. Initial and final cuts of the dynamics

action of p_1 is an action by P from global state s, and that its last action is by P' with end global state s'. Similarly p_2 starts at s by an action by P' and ends in s' with an action by P in the following sum,

$$\alpha'(T) = \bigcup\{(p_I(s), q_I(s), p_O(s'), q_O(s'))/\text{there is a 2-path between } p_1 \text{ and } p_2\}$$

This actually generalises the denotational abstraction α of Section 5.

If you think that P or P' may fail and give $\perp$ as a result then what we wrote as $((P,u),(P',v),(P,x),(P',y)) \in \alpha'(T)$ (or $(u,v) \to (x,y)$ as a shortcut) is now $((P,u),(P,x)) \in \alpha(T)$ (or $(u,\perp) \to (x,\perp)$ as a shortcut).

Both of these control flow abstractions can be represented as follows. If we look at the semantics as real geometric shapes, we are looking at relations between the source and target of the semantics. We call source the initial "cut" of the dynamics, and target the final "cut" of the dynamics, as shown in Figure 3.

As shown also in Figure 3, the relation induced by the dynamics is clearly a relation between two graphs, one is called the input graph, the other, the output graph.

Formally, the input and output values are nodes of a graph that we will call the *compatibility graph* $S_{\mathbb{Z}} = (V,E)$ defined as follows (see Figure 8 for a picture of $S_{[1,M]\cap\mathbb{Z}}$).

— its set of vertices is $V = \{P\} \times \mathbb{Z} \cup \{Q\} \times \mathbb{Z}$,

— its set of edges is $E = \{(v_1,v_2)/v_1 = (P,r), v_2 = (Q,s)\}$ with the obvious boundaries.

And then, it is easy to check that all possible input and output graphs are subgraphs of $S_{\mathbb{Z}}$. The input graph, the output graph and the relation between these is called the *specification graph.*

8. Some General Properties of HDA

To speak about "geometric" properties of paths we will be needing in the sequel, we need to change our point of view on the objects (HDA) we are manipulating.

Definition 2. A (unlabeled) *higher dimensional automaton* (HDA) is a R-module M with two gradings associated to two boundary operators ∂_0 and ∂_1, that is, consists in:

— a decomposition: M$=\sum_{p,q\in\mathbb{Z}} M_{p,q}$, such that

$$\forall n, \left(\sum_{p+q=n} M_{p,q}\right) \cap \left(\sum_{r+s\neq n} M_{r,s}\right) = 0$$

— two differentials ∂_0 and ∂_1, compatible with the decomposition, giving M a structure of bicomplex:

$$\partial_0 : M_{p,q} \longrightarrow M_{p-1,q}$$

$$\partial_1 : M_{p,q} \longrightarrow M_{p,q-1}$$

$$\partial_0 \circ \partial_0 = 0, \quad \partial_1 \circ \partial_1 = 0, \quad \partial_0 \circ \partial_1 + \partial_1 \circ \partial_0 = 0$$

The connection between this definition and the one we had before is stated in the following lemma,

Lemma 2. Let M be a semi-regular HDA. Then $\underline{M}$ defined as,

— $\underline{M}_{p,q}$ is the free R-module generated by $M_{p,q}$,
— $\partial_0 = \sum_{i=0}^{i=p+q-1}(-1)^i d_i^0$,
— $\partial_1 = \sum_{i=0}^{i=p+q-1}(-1)^i d_i^1$.

is a general HDA.

Proof. We just check here that $\partial_0 \circ \partial_0 = 0$. The other two equations can be verified in a similar manner. Let $x \in M_{p,q}$,

$$\begin{aligned}
\partial_0 \circ \partial_0(x) &= \sum_{i=0}^{i=p+q-2}\sum_{j=0}^{j=p+q-1}(-1)^{i+j} d_i^0 \circ d_j^0(x) \\
&= \sum_{0\leq i<j\leq p+q-1}(-1)^{i+j} d_i^0 \circ d_j^0(x) + \sum_{p+q-2\geq i\geq j\geq 0}(-1)^{i+j} d_i^0 \circ d_j^0(x) \\
&= \sum_{0\leq i<j\leq p+q-1}(-1)^{i+j} d_{j-1}^0 d_i^0(x) + \sum_{p+q-2\geq i\geq j\geq 0}(-1)^{i+j} d_i^0 \circ d_j^0(x) \\
&= \sum_{0\leq J\leq I\leq p+q-2} -(-1)^{I+J} d_I^0 d_J^0(x) + \sum_{p+q-2\geq i\geq j\geq 0}(-1)^{i+j} d_i^0 \circ d_j^0(x) \\
&= 0
\end{aligned}$$

□

Paths of M can be easily defined in $\underline{M}$. For instance, a (sequential) path from state $\alpha \in M_{p,-p}$ to state $\beta \in M_{p+k,-p-k}$ is nothing but a sequence of 1-transitions that we can picture as follows,

$$\begin{tikzcd}
 & & p_1 \in M_{p+1,-p} \arrow[r, "d_0^0"] \arrow[d, "d_0^1"'] & \alpha \\
 & p_1 \in M_{p+2,-p-1} \arrow[r, "d_0^0"] & = & \\
 & \vdots & & \\
p_k \in M_{p+k,-p-k+1} \arrow[r, "d_0^0"] \arrow[d, "d_0^1"'] & = & & \\
\beta & & &
\end{tikzcd}$$

thus verifying that the end of p_i is the beginning of p_{i+1} and that the beginning of p_1 is α and the end of p_k is β. Let us call $P_1^{\alpha,\beta}$ the set of such paths (called 1-paths of paths of dimension 1 since it only involves sequential executions) from state α to state β.

Before looking at these objects, and to their higher-dimensional analogues, we have to restrict to a convenient case,

Lemma 3. A HDA M is such that $M = \bigoplus_{p,q} M_{p,q}$ (meaning that $(p,q) \neq (p',q') \Rightarrow M_{p,q} \cap M_{p',q'} = 0$) if and only if for all $p \neq q$, $M_{p,-p} \cap M_{q,-q} = 0$. Such HDA will be called acyclic.

Proof. Suppose that there are two distinct pairs of indexes (p,q) and (p',q') such that $M_{p,q} \cap M_{p',q'} \neq \emptyset$. Let $A \in M_{p,q} \cap M_{p',q'}$. Then let $x = d^0_{p+q-1} \circ \cdots \circ d^0_1 \circ d^0_0(A)$. As $p+q = p'+q'$ (one of the axioms of HDA) $x \in M_{q,-q}$ and $x \in M_{q',-q'}$. As $p+q = p'+q'$ and $(p,q) \neq (p',q')$, we have $q < q'$ or $q' < q$ but not $q = q'$. □

Acyclic HDA M have the property that any state is in a unique submodule $M_{p,-p}$ of M and that any path from a point $\alpha \in M_{p,-p}$ and $\beta \in M_{k+1+p,-p-k-1}$ has length k.

There is now a correspondance between these paths and a computable object in $\underline{M}$, for M acyclic,

Lemma 4. The R-module generated by elements $x = (x_1, \cdots, x_k)$ such that there is a $\lambda \in R$ with,

— $x_i \in M_{p+i,-p-i+1}$,
— $\partial_0(x) = \partial_1(x) + \lambda(\alpha - \beta)$,

is isomorphic to the R-module generated by $P_1^{\alpha,\beta}$.

Proof. Let N be the first module defined in the lemma and define the linear function,

$$\begin{array}{rlcl} f: & R-Mod(P_1^{\alpha,\beta}) & \to & N \\ & (x_1,\cdots,x_k) & \to & (x_1,\cdots,x_k) \end{array}$$

First we prove that the image of f is indeed in N. Let $y = \sum_{i=1}^m \lambda_i (x_1^i, \cdots, x_k^i)$ with $(x_1^i, \cdots, x_k^i) \in P_1^{\alpha,\beta}$ be an element of $R - Mod(P_1^{\alpha,\beta})$. Notice first that for any path $(x_1, \cdots, x_k) \in P_1^{\alpha,\beta}$ we have, if $x = f(x_1, \cdots, x_k)$,

$$\begin{array}{rcl} \partial_0(x) - \partial_1(x) & = & \alpha + \partial_0(x_2) - \partial_1(x_1) + \cdots + \partial_0(x_k) - \partial_1(x_{k-1}) - \beta \\ & = & \alpha - \beta \end{array}$$

Therefore,

$$\begin{array}{rcl} \partial_0(y) - \partial_1(y) & = & \sum_{i=1}^m \lambda_i (\partial_0 - \partial_1)(f(x_1^i, \cdots, x_k^i)) \\ & = & (\sum_{i=1}^m \lambda_i)(\alpha - \beta) \end{array}$$

f is an isomorphism since it is defined as the identity map. □

Nevertheless, we do not have $R - Mod(P_1^{\alpha,\beta})$ isomorphic to the R-module $\underline{P}_1^{\alpha,\beta}$ defined as being the sub-R-module of $\underline{M}_{p+1,-p} \times \cdots \underline{M}_{p+k,-p-k+1}$ of x such that $\partial_0(x) = \partial_1(x) + \lambda(\alpha - \beta)$ since these are generated by all paths, even the undirected ones, between α and β.

All this can be generalised in higher dimensions in a easier way. Intuitively, a n-path is a path where n processors are acting together asynchronously. Basically, these will be

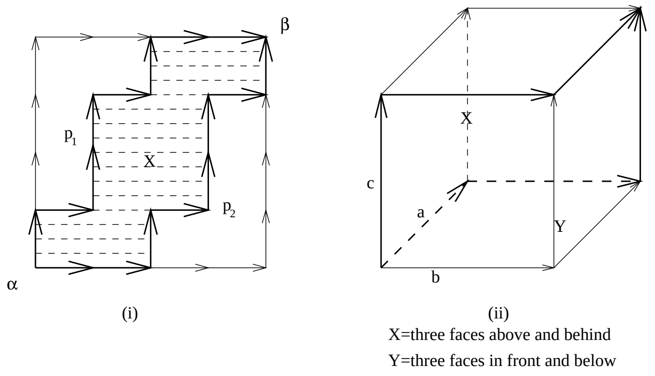
β
p_1
X
p_2
α
(i)
X
c
a
Y
b
(ii)
X=three faces above and behind
Y=three faces in front and below

Fig. 4. A surface within two paths p_1 and p_2, and a hypersurface (the cube) between two surfaces X and Y

some kind of sequences of n-transitions. These can be defined formally by considering $\underline{M}$ directly, because from dimension 2 on, there is no need to direct surfaces.

Definition 3. Let $n \geq 1$ and M be a acyclic HDA. Let p_1 and p_2 be two $(n-1)$-paths between two $(n-2)$-paths α and β (or if $n=1$, p_1 and p_2 are just two states).
Then the R-module of n-paths between p_1 and p_2 is the R-module composed of elements x such that there exists $\lambda \in R$ with,

$$\partial_0(x) = \partial_1(x) + \lambda(p_1 - p_2)$$

This R-module is named $\underline{P}_n^{p_1,p_2}(M)$.

This means that, supposing $p_i = (p_i^1, \ldots, p_i^k)$ and $p_i^1 \in M_{n-1+s,-s}$, its elements $x \in P_n^{p_1,p_2}(M)$ are $x = (x^1, \ldots, x^{k-1})$ such that

— $x^s \in \underline{M}_{n+s,-s}$,
— $\partial_0(x^{i+1}) - \partial_1(x^i) = p_1^{i+1} - p_2^{i+1} \in \underline{M}_{n+i,-i-1}$,
— $\partial_0(x^1) = p_1^1 - p_2^1$,
— $-\partial_1(x^k) = p_1^k - p_2^k$.

The basic idea is that a n-path (or hypersurface of dimension n) is enclosed at each time i within the ith stage of p_1 and the ith stage of p_2, as shown in Figure 4.

Now we can define what it is for a HDA to be wait-free. Intuitively, a program composed of n processes in parallel is wait-free if and only if the program terminates with a partial result even if t ($t \leq n-1$) processes fail. In the case of two processors, it is easy to see that this can be characterized in a geometric manner (see Figure 5). Basically, mutual exclusions are not wait-free whereas asynchronous executions are.

This can be characterized in terms of n-paths,

Definition 4. A HDA M is wait-free if and only if for all $2 \leq t \leq n$, for all $(t-1)$-paths p and q ($p \neq q$) between α and β, $\underline{P}_t^{p,q} \neq 0$.

This means that all executions are in fact part of asynchronous executions of t ($t \leq n$) processes ($2 \leq t \leq n$). Another way of seeing that is to take p to be any execution and q an execution in which the program of some process P_j is entirely executed before

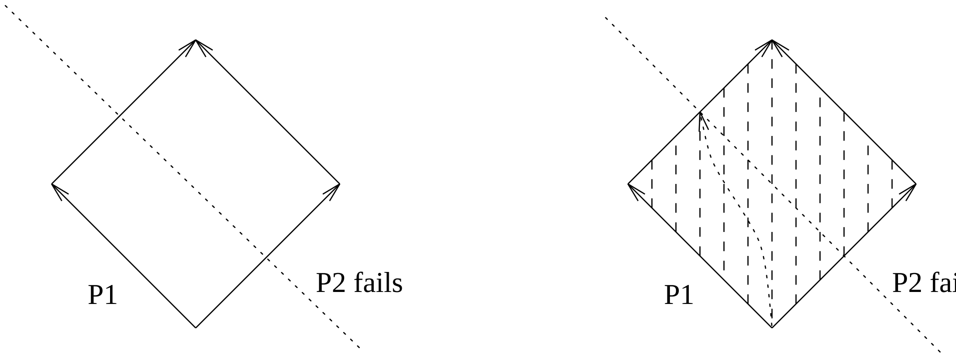

Fig. 5. Geometry of non-wait-free with respect to wait-free programs

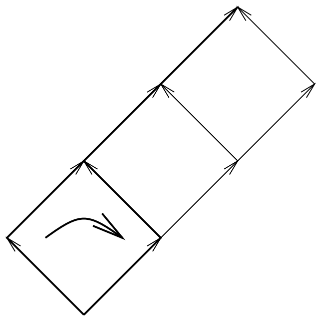

Fig. 6. Elementary reschedulings of actions

the programs of P_i, $i \neq j$. Then M wait-free means that the execution can always be reordered (by the permutations of actions specified by an element of $\underline{P}_t^{p,q}$, see Figure 6) so that P_j is executed first. Therefore, if P_j is the only non-faulty process, it will terminate even if the others fail at some point.

Then the following characterization is useful,

Proposition 1. Let M be a connected acyclic HDA of dimension n. Then M is wait-free if M is $(n-1)$-connected, or if M is such that all its homology groups $H_k(M) = Ker(\partial_0 - \partial_1)/Im(\partial_0 - \partial_1)$ $(k \leq n-1)$ are 0.

Proof. The implication between the last two statements is Hurewicz theorem (May67; GZ67).
We prove now that M is wait-free if all its homology groups up to dimension $n-1$ are trivial.
Suppose M is not wait-free. Then there exist two $(t-1)$-paths p and q $(p \neq q)$ between some α and some β such that $\underline{P}_t^{p,q} = 0$ $(2 \leq t \leq n)$, by Definition 4. We show now that $p - q \in H_t(M)$ hence (as $p - q \neq 0$) $H_t(M) \neq 0$.

First, $p-q \in Ker(\partial_0 - \partial_1)$. To prove this write $p=(p_1,\cdots,p_k)$ and $q=(q_1,\cdots,q_k)$ (acyclicity implies that p and q have the same length). Then,

$$\begin{array}{rcl}(\partial_0-\partial_1)(p-q) & = & (\partial_0-\partial_1)(p)-(\partial_0-\partial_1)(q) \\ & = & \alpha-\beta-(\alpha-\beta) \\ & = & 0\end{array}$$

Then we prove that $p-q \notin Im(\partial_0-\partial_1)$.
Suppose the contrary, i.e. $p-q=(\partial_0-\partial_1)(A)$ for some A. This proves $A \in \underline{P}_t^{p,q}$ hence a contradiction since $\underline{P}_t^{p,q}=0$. □

Basically this says that M is wait-free if for any two t-paths p and q between α and β, we have p and q homotopic. This also implies that homotopic t-paths (or t-schedules (Gou95)) start at the same global state and end at the same global state, therefore, t-schedules characterize the possible outcomes of a distributed computation.

We need some general statements about k-connectedness, which will be useful in the sequel.

Lemma 5. Let A and B be two connected and k-connected shapes such that $A \cap B$ is connected and k-connected ($k \geq 1$). Then $A \cup B$ is connected and k-connected.

Proof. By Hurewicz theorem, being connected and k-connected is equivalent to being connected and simply-connected and having all its homology groups up to k being trivial. By Seifert/Van Kampen's theorem (GZ67), $A \cup B$ is connected and simply-connected. We can now prove k-connectedness of $A \cup B$ by looking only at its homology groups. We use the Mayer-Vietoris exact sequence (ML63),

$$\cdots \longrightarrow H_k(A\cap B) \longrightarrow H_k(A)\oplus H_k(B) \longrightarrow H_k(A\cup B) \longrightarrow H_{k-1}(A\cap B) \longrightarrow \cdots$$

All terms appearing in this exact sequence (at the right of the first $\cdots$) are equal to 0 except for $H_k(A\cup B)$, $H_{k-1}(A\cup B)$ etc. that we do no know yet. The exactness of the sequence above force them to be also equal to 0. □

9. Some General Properties of the Second Semantics

To be able to understand what the language we are considering can compute, we need an accurate picture of the semantics of every term (at least modulo some kind of "directed homotopy" (Gun94)).

First, we need a general remark on the SOS-style definition of programming languages. Basically, when defining a language by SOS rules we only consider those states which are reachable from the initial state $i=(\{P,Q\},\rho)$ by an increasing path. Therefore, if the initial state is in $M_{0,0}$ and if x is another state of M then there is at least one 1-path $p=(p_1,\cdots,p_k)$ from i to x therefore $M_{k,-k}$ contains x (others might as well contain x though). This also implies that the semantics of any term is a connected space.

Lemma 6. Both the concrete HDA semantics of Section 6 and the abstract control flow semantics of any term from any environment are acyclic.

Proof. Define the size σ of a state $(\{P,Q\},\rho)$ as the pair of the numbers of ; plus the numbers of *case* in the strings P and P'. Obviously $s = s' \Rightarrow \sigma(s) = \sigma(s')$. Then it is easy to show on the semantics that for all actions a

$$(\{P, Q\}, \rho) \xrightarrow{a} (\{P', Q'\}, \rho')$$

we have $\sigma(\{P, Q\}, \rho) = \sigma(\{P', Q'\}, \rho') + (1, 0)$ or $\sigma(\{P, Q\}, \rho) = \sigma(\{P', Q'\}, \rho') + (0, 1)$ with the pointwise addition on $\mathbb{N}^2$. Therefore the execution of a program looks like an unfolding (because of the actual value of ρ) of a sub-grid of $\mathbb{N}^2$. This means that all paths from the initial state to any state x has the same length and that a state cannot belong to $M_{p,-p}$ and $M_{q,-q}$ with $p \neq q$. Hence M is acyclic by Lemma 3. □

Then,

Lemma 7. There is always a unique solo execution of P (respectively of P') from any global state $(\{P, Q\}, \rho)$. It is denoted $p_{(\{P,Q\},\rho)}$ (respectively $q_{(\{P,Q\},\rho)}$).

Proof. This amounts to proving that each process written in our language is purely deterministic. We prove that given a state $s = (\{P, Q\}, \rho)$ there cannot be two 1-transitions a and b of P such that,

$$\begin{tikzcd} & s' \\ s \arrow[ur, "b"] \arrow[dr, "a"'] & \\ & s'' \end{tikzcd}$$

with $s' = (\{P', Q\}, \rho')$ different from $s'' = (\{P'', Q\}, \rho'')$. Looking at the standard semantics, we have two cases,

— $a \neq case$ and $b \neq case$. Then $a = b$ and $P' = P''$. Each of the rules (*update*), (*scan*) and (*calc*) imply $\rho' = \rho''$ thus $s' = s''$.

— $a = b = case$. Then,

$$P = \left(\begin{array}{l} case\ (u_1 \dots u_k)\ of \\ (a_1^1 \dots a_k^1)\ :\ P_1 \\ \cdots \\ (a_1^n \dots a_k^n)\ :\ P_n \\ default\ :\ P \end{array} \right); R$$

and $P' = P'' = P_k; R$ (for some k) or $P' = P'' = P; R$. Furthermore the rule (*case*) implies that $\rho' = \rho''$ hence $s' = s''$.

□

The HDA semantics still allows us to speak of solo executions, but also allows us to reason about 2-processor executions (or here, global executions).

Theorem 1. The abstract control flow semantics of Section 7 of any term, for any initial environment, is wait-free.

To prove this, we first need a lemma describing the intersection of the semantics of two programs.

Lemma 8. Let P_1 and P_2 be two programs, ρ_1 and ρ_2 two contexts. Then,

$$[\![P_1]\!]\rho_1 \cap [\![P_2]\!]\rho_2 = \cup_{(x,\rho)\in[\![P_1]\!]\rho_1\cap[\![P_2]\!]\rho_2}[\![x]\!]\rho$$

Proof. By Lemma 7, given any state $s = (\{a; P', b; Q'\}, \rho)$ there are unique solo executions by P or by Q from it, hence, looking at the rules $(no\ interference)$ and $(interference)$ we have two cases,

— $(a, b) \neq (scan, update)$ and $(a, b) \neq (update, scan)$ then there is a unique 2-transition t that can be fired between the solo executions p_s and q_s. We note

$$s \xrightarrow{t} s'$$

— $(a, b) = (scan, update)$ or $(a, b) = (update, scan)$ then two 2-transitions t_1 and t_2 can be fired between p_s and q_s. We also note (for $i = 1, 2$),

$$s \xrightarrow{t_i} s'$$

Using this new notation (generalizing the notation on 1-transitions) the proof goes as follows.
Let $(x, \rho) \xrightarrow{t} (x', \rho')$ be a transition (of any dimension) in $[\![P_1]\!]\rho_1 \cap [\![P_2]\!]\rho_2$, then $(x, \rho) \in [\![P_1]\!]\rho_1 \cap [\![P_2]\!]\rho_2$ and $t \in [\![x]\!]\rho$.
Reciprocally, let $(x, \rho) \in [\![P_1]\!]\rho_1 \cap [\![P_2]\!]\rho_2$ and $t \in [\![x]\!]\rho$ and $(y, \rho') \xrightarrow{t} (z, \rho'')$. There is also a path from (x, ρ) to (y, ρ'). As $(x, \rho) \in [\![P_1]\!]\rho_1$ (respectively $(x, \rho) \in [\![P_2]\!]\rho_2$) then there is a path from (P_1, ρ_1) (respectively (P_2, ρ_2)) to (x, ρ) hence to (y, ρ'). Therefore $t \in [\![P_1]\!]\rho_1$ and $t \in [\![P_2]\!]\rho_2$. $\square$

Then the proof of Theorem 1 is as follows,

Proof. We prove that for all ρ, $[\![x]\!]_c^\rho$ is $(n-1)$-connected and even more, is contractible, by induction on the size $\sigma(x)$. By and Proposition 5 this will entail that for all programs x, its semantics is wait-free[‡].
The ground case is $\sigma(x) = 0$ then $[\![x]\!]_c^\rho = \emptyset$ which is contractible.
Now take x such that $\sigma(x) > 0$ and suppose the result hold for all x' started in any environment ρ', with $\sigma(x') < \sigma(x)$.
$\sigma(x) > 0$ hence we have the following three case,
(1) $x = \{a; P', \epsilon\}$.
(2) $x = \{\epsilon, b; Q'\}$.
(3) $x = \{a; P', b; Q'\}$.

In cases (1) and (2), $[\![x]\!]_c^\rho$ is the union of the segment $x \xrightarrow{a} x'$ and of $[\![x']\!]_c^{\rho'}$. $\sigma(x') < \sigma(x)$ hence $[\![x']\!]_c^{\rho'}$ is contractible. The segment t is also contractible. The intersection of both spaces is the point x'. As $[\![x]\!]_c^\rho$ is connected we can choose any base point for homotopy groups as they are all isomorphic. Choose as base point x'. Any (even higher-dimensional)

[‡] To relate to the ordinary notion of wait-freeness, we only need to prove that $\alpha_c(x)$ is $(n-1)$-connected, since the only important thing to know is if we can reschedule actions, no matter the actual values in the memory are.

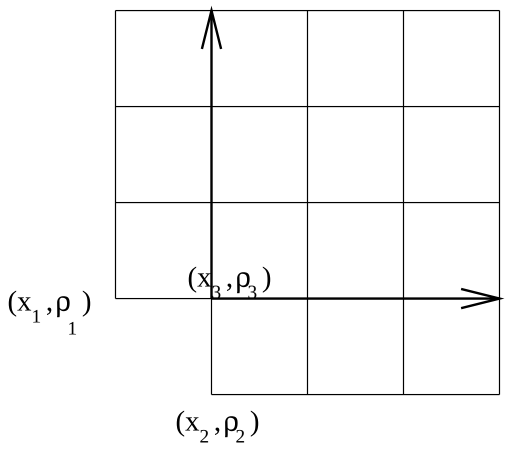

Fig. 7. the relative configuration between $[\![x_1]\!]\rho_1$, $[\![x_2]\!]\rho_2$ and $[\![x_3]\!]\rho_3$

loop is a composition of loops in t or in $[\![x']\!]_c^{\rho'}$, each of which is contractible to the base point. Hence, every loop is homotopic to x' in $[\![x]\!]_c^{\rho}$, and $[\![x]\!]_c^{\rho}$ is contractible.
Case (3) is more complex. We have *a priori* two subcases,

(i) $(a, b) \neq (scan, update)$ and $(a, b) \neq (update, scan)$.
(ii) $(a, b) = (scan, update)$ or $(a, b) = (update, scan)$.

In case (i), the beginning of the dynamics at (x, ρ) looks like,

$$\begin{tikzcd} & (x_1, \rho_1) \arrow[dr, "b"] & \\ (x, \rho) \arrow[ur, "a"] \arrow[dr, "b"'] & t & (x_3, \rho_3) \\ & (x_2, \rho_2) \arrow[ur, "a"'] & \end{tikzcd}$$

Now $[\![x]\!]_c^{\rho}$ is the union of t (a square, hence contractible), of $[\![x_1]\!]_c^{\rho_1}$ and of $[\![x_2]\!]_c^{\rho_2}$. Both $[\![x_1]\!]_c^{\rho_1}$ and $[\![x_2]\!]_c^{\rho_2}$ are contractible by the induction hypothesis since $\sigma(x_1) < \sigma(x)$ and $\sigma(x_2) < \sigma(x)$. Now, by Lemma 8, the intersection of $[\![x_1]\!]\rho_1$ and $[\![x_2]\!]\rho_2$ is a union of some $[\![x']\!]\rho'$. $[\![x_3]\!]\rho_3$ is one of these since it is obviously in both $[\![x_1]\!]\rho_1$ and $[\![x_2]\!]\rho_2$. But we know that the execution is all tied up within sub-grids of $\mathbb{N}^2$ (when you do not look at the environments). So $[\![x_3]\!]\rho_3$ has to be the maximal element of the set of all these $[\![x']\!]\rho'$, hence should contain them all (look at Figure 7).
By the induction hypothesis, $[\![x_3]\!]_c^{\rho_3}$ is contractible, so $U = [\![x_1]\!]_c^{\rho_1} \cup [\![x_2]\!]_c^{\rho_2}$ is contractible (Lemma 5). Now, U and t have only in common two connected segments, so once again, as t is contractible, $[\![x]\!]_c^{\rho} = t \cup U$ is contractible.
Case (ii) is just about the same. The only difference is that the beginning of the control flow semantics is the union of a square, of $[\![x_1]\!]_c^{\rho_1}$ and of $[\![x_2]\!]_c^{\rho_2}$ such that,

— by the induction hypothesis, $[\![x_1]\!]_c^{\rho_1}$ is contractible,
— by the induction hypothesis, $[\![x_2]\!]_c^{\rho_2}$ is contractible,
— the square t is contractible,

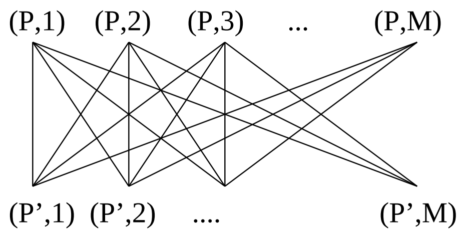

Fig. 8. The input graph for values in $[1, M] \cap \mathbb{Z}$.

— the intersection of $[\![x_1]\!]\rho_1$ and $[\![x_2]\!]\rho_2$ is a point.

Hence $[\![x]\!]_c^\rho$ is contractible as the amalgamated sum of two contractible shapes ($t \cup [\![x_1]\!]_c^{\rho_1}$ and $[\![x_2]\!]_c^{\rho_2}$) above a contractible subshape. $\square$

10. Geometric properties and impossibility results

Specification graphs represent the relation computed by programs written in our wait-free language. Conversely, given a binary relation, can we determine whether it can be implemented in our language (that is, whether it is a wait-free binary relation or whether it is the "denotational" semantics of some program in our language)? The answer is yes, and could be proved as a particular case of a general theorem by M. Herlihy and N. Shavit (HS93). The criterion in our case is as follows. Suppose that P and P' ran alone (i.e. with the other process not being fired in parallel) are the identity functions on their inputs, and that the allowed initial states are such that $\rho(x) = \rho(y) = \bot$, then,

Theorem 2. Let $\{e_1, \ldots, e_k\}$ be the image of a segment $e = ((P, u), (P', v'))$ of the input graph under the relation Δ, i.e. the set of segments e' such that $e \Delta e'$. Then $e_1, \ldots, e_k$ is a path from (P, u) to (P', v') in the output graph.

Proof. We do know pretty much of the shape of the dynamics in our little language (thanks to the detailed proof of wait-freeness, Theorem 1). The only states at which we have a choice between two behaviours are states like,

(1) $(\{scan_P; P', update_{P'}; Q'\}, \rho)$,
(2) $(\{update_P; P', scan_{P'}; Q'\}, \rho)$.

At each of these, we may unfold the "filled-in subgrid of $\mathbb{N}^2$" which would normally be the shape of the remaining of the dynamics into two of these (as you choose between the upper face t_1 or the lower face t_2 in the semantics). Therefore, at each of these "critical points" we have a kind of a "paper clip" shape in the control flow semantics (see Figure 9). These shapes may appear anywhere on the underlying grid. This might give "multiple paper clip" as in Figure 10.

A multiple paper clip separates out layers which we can index as in Figure 10 from level 1 to level n going from the P edge to the P' edge (the index is based on the size of the P' process). Looking at the final cuts of the dynamics we see that each cut is a segment, since the dynamics is wait-free and 2-dimensional and that each of the segments correspond biunivoquely (the layers are all distinct) to the different possible final environments, and to the homotopy classes of 1-paths (see Figure 10).

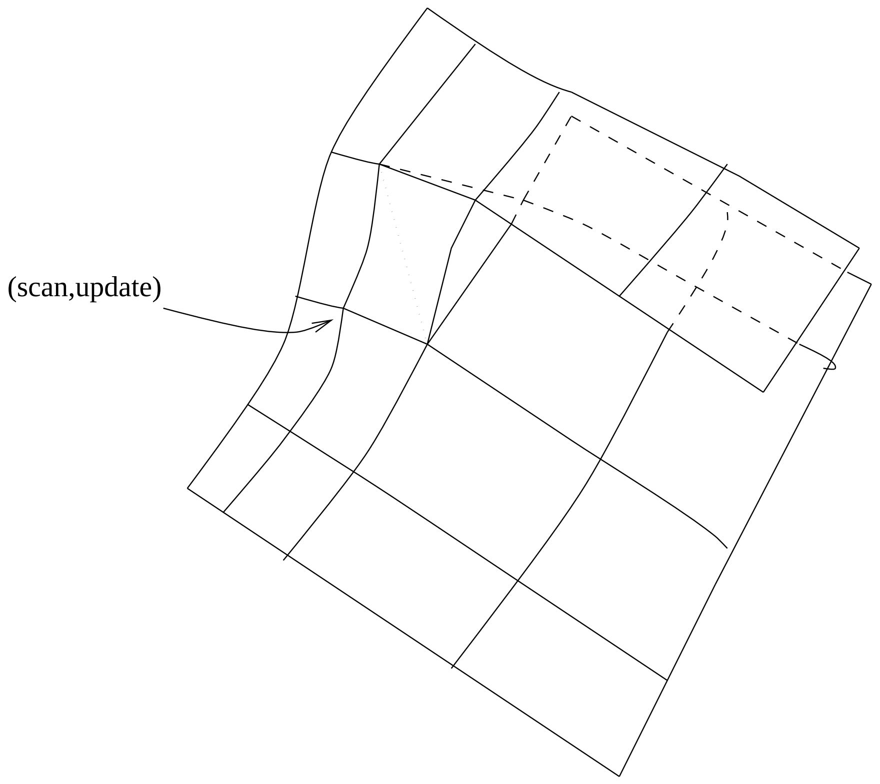

Fig. 9. The “paper clip” shape

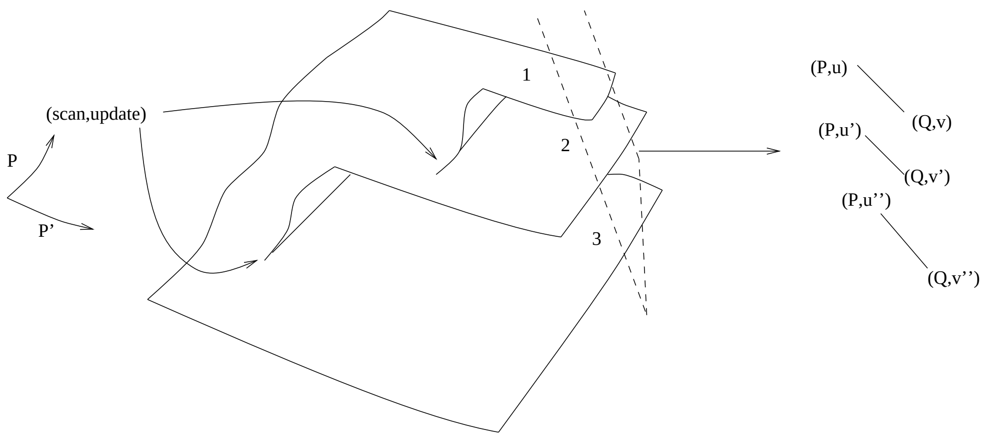

Fig. 10. Multiple paper clip

Now we have characterized the control flow by exhibiting the essential schedulers (the homotopy classes of 1-paths(Gou95)), we have to see how the contents of the environment is changed when looking at different schedules (i.e. when looking at different histories of interactions between P and P').
Let us carry on with our assumption that we have n different layers labelled from 1 to n. Layer $i+1$ is separated from layer i by a critical point c_i (with environment ρ_i) of type n_i being 1 or 2. The semantic rule dealing with *scan* and *update* operations shows us that the action on the environment is as follows,

(1) $n_i = 1$. Then the new environment ρ',

   (a) at the start of layer $i+1$ (execution of $scan_P$ before $update_{P'}$) is such that $\rho'(x) = \rho_i(x)$, $\rho'(y) = \rho_i(v')$, $\rho'(u) = \rho_i(u)$, $\rho'(v) = \rho_i(y)$, $\rho'(u') = \rho_i(u')$ and $\rho'(v') = \rho_i(v')$,

   (b) at the start of layer i (execution of $scan_P$ before $update_{P'}$) is such that $\rho'(x) = \rho_i(x)$, $\rho'(y) = \rho_i(v')$, $\rho'(u) = \rho_i(u)$, $\rho'(v) = \rho_i(v)$, $\rho'(u') = \rho_i(u')$ and $\rho'(v') = \rho_i(v')$.

(2) $n_i = 2$. Just exchange the local variables of P and P' in subcases (a) and (b).

Therefore, if we call (α_i, β_i) the segment which is the final cut semantics for layer i, we have the following equations,

(1) if $n_i = 1$ then the difference between the environments of layer i and layer $i+1$ can only reside in the variables local to P and as there is no other $(scan, update)$ conflict after c_i, there are only local computations to P and P' remaining, therefore $\beta_{i+1} = \beta_i$: only the variables of P may change in layer $i+1$ with respect to layer i (by induction on the information flow semantics),

(2) if $n_i = 2$ then the difference between the environments of layer i and layer $i+1$ can only reside in the variable local to P' (by looking at the information flow semantics), therefore $\alpha_{i+1} = \alpha_i$.

Therefore, the n segments in the final cut semantics form a connected graph. As P' cannot modify P's variables nor P can modify P''s variables (reasoning on the information flow semantics), two of the points in this connected graph are the solo executions of P and P' respectively (corresponding to the schedules $P; P'$ and $P'; P$ respectively). $\square$

This geometric condition is satisfied for the pseudo-consensus relation as one can see by looking at the specification graph of Figure 11.

The situation is not quite the same with binary consensus (Figure 12). An easy inspection shows that the image of the segment $((P, 0), (P', 1))$ is a set of two disconnected segments, thus violating the result of Theorem 2. Therefore, binary consensus cannot be implemented in a wait-free manner. The intuition behind this result is quite simple. Consensus requires that a process can tell whether it is the first or last to choose, because otherwise there is no way to be sure that the two processes will agree on any value. This means it needs a synchronization, a break of the connexity of the cuts of the dynamics (Gou96). This is of course impossible in a wait-free language.

Similarly, $\underline{\text{if the input values are given locally to the processes}}$ as we supposed in Theorem 2, parallel or (or ordered binary consensus, see the specification graph, Figure 13)

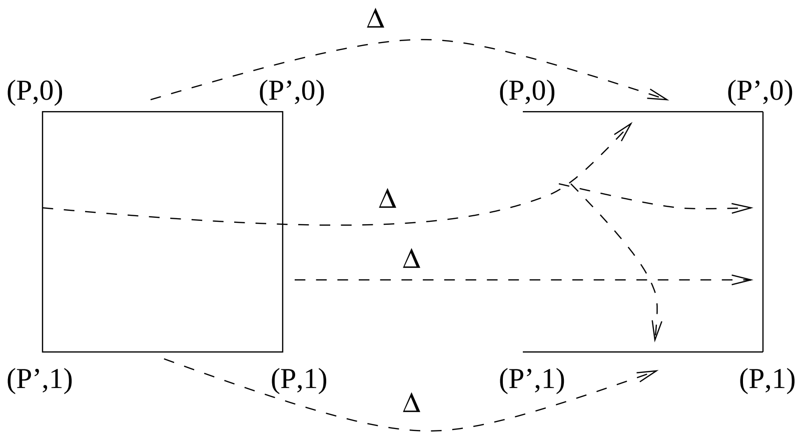

Fig. 11. The specification of the binary pseudo-consensus.

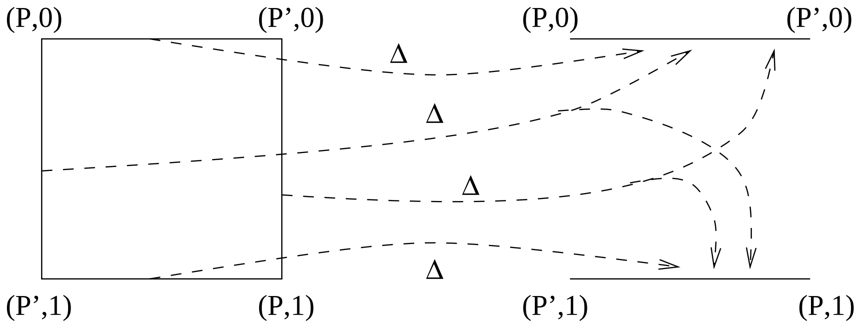

Fig. 12. The specification of the binary consensus.

cannot be implemented in a wait-free manner. There is though a wait-free solution for parallel or if the input is stored in the shared memory right from the beginning:

```
Prog = P | Q
P    = update;                  Q = update;
       scan;                        scan;
       case v of                    case u of
          1 : u = 1; update            1 : v = 1; update
          default : update             default : update
```

The connectivity condition is indeed preserved throughout the execution of parallel or. But starting with a non-empty environment implies that there might just be no relation between solo executions and 2-schedules.

11. Geometric properties: the reciprocal

To complete our "full abstraction" theorem, we start off with a relation Δ between input/output values and pairs of inputs/pairs of outputs.

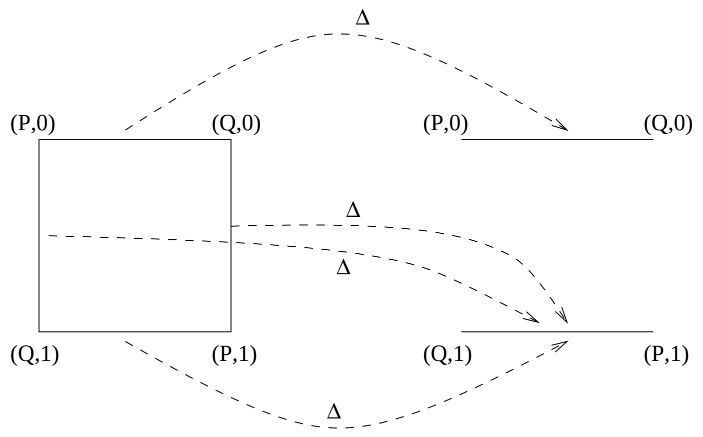

Fig. 13. The specification of parallel or.

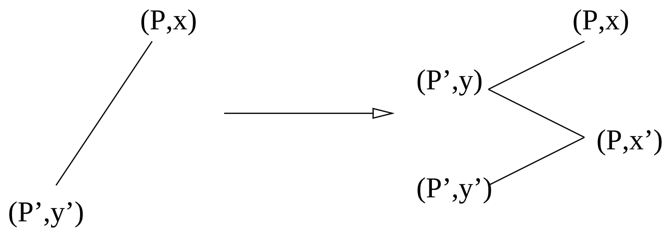

Fig. 14. Subdivision of a segment into three segments.

11.1. *Usual Case*

We suppose here that all paths image by Δ of any segment of the input graph are made of distinct segments (one should say, oriented segments, as we will see later on). We can also suppose here that Δ restricted to vertices is the identity relation.

11.1.1. *Subdivision of a segment into three segments* The program $Prog = P[\epsilon] \mid P'[\epsilon]$ with P and P' defined below (being programs with one hole [] in which we can plug any other program) implements the specification graph of Figure 14 (the segments not being pictured are mapped onto themselves).

$$\begin{array}{rcl} P & = & update; \\ & & scan; \\ & & case\ (u,v)\ of \\ & & \ (x,y')\ :\ u=x'; update; [] \\ & & \ default\ :\ update \end{array} \qquad \begin{array}{rcl} P' & = & update; \\ & & scan; \\ & & case\ (u,v)\ of \\ & & \ (x,y')\ :\ v=y; update; [] \\ & & \ default\ :\ update \end{array}$$

Proof. Using the semantics, we have the following three possible 1-schedules (up to homotopy), since the only possible interactions are between the *scan* and *update* statements,

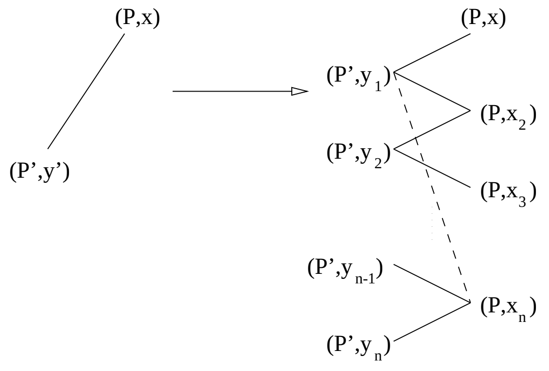

Fig. 15. Subdivision of a segment into a path.

(i) Suppose the *scan* operation of P is completed before the *update* operation of P' is started: P does not know y so it chooses to write x. *Prog* ends up with $((P,x),(P',y))$.
(ii) Symmetric case: *Prog* ends up with $((P,x'),(P',y'))$.
(iii)The *scan* operation of P is after the *update* of P' and the *scan* of P' is after the *update* of P. *Prog* ends up with $((P,x'),(P',y))$.

The semantics of this program from the segment (x,y') has the "multiple paper clip" shape of Figure 10. □

Example 1. The binary pseudo-consensus whose specification graph is given in Figure 11 is implemented by this program with $x=0$, $x'=1$, $y=0$, $y'=1$.

11.1.2. *Subdivision of a segment into a path* The program

$$Prog = P(x_1,y_1,\cdots,x_n,y_n) \mid P'(x_1,y_1,\cdots,x_n,y_n)$$

with P and P' defined below, implements the specification graph of Figure 15.

$$\begin{array}{rcl} P(x_1,y_1,\cdots,x_n,y_n) & = & P(x_1,y_1,x_n,y_n) \\ & & \quad [P(x_n,y_{n-1},\cdots,x_2,y_1)] \\ P'(x_1,y_1,\cdots,x_n,y_n) & = & P'(x_1,y_1,x_n,y_n) \\ & & \quad [P'(x_n,y_{n-1},\cdots,x_2,y_1)] \end{array}$$

where $P(x_1,y_1,x_n,y_n) \mid P'(x_1,y_1,x_n,y_n)$ is the program of last section with $x=x_1$, $y=y_1$, $x'=x_n$ and $y'=y_n$.

Proof. The idea is to subdivide the segment (x_1,y_n) in a recursive manner (see Figure 15). First subdivide (x_1,y_n) into $\{(x_1,y_1),(x_n,y_1),(x_n,y_n)\}$ by using the program $P(x_1,y_1,x_n,y_n) \mid P'(x_1,y_1,x_n,y_n)$. Then subdivide recursively (x_n,y_1) into the path of length $n-1$ $(x_n,y_{n-1},\ldots,x_2,y_1)$ using $P(x_n,y_{n-1},\ldots,x_2,y_1) \mid P'(x_n,y_{n-1},\ldots,x_2,y_1)$. *Prog* works since (as all the segments (x_i,y_i) are distinct) there is no interference between $P(x_1,y_1,x_n,y_n)$ and $P'(x_n,y_{n-1},\ldots,x_2,y_1)$ nor between $P'(x_1,y_1,x_n,y_n)$ and $P(x_n,y_{n-1},\ldots,x_2,y_1)$. □

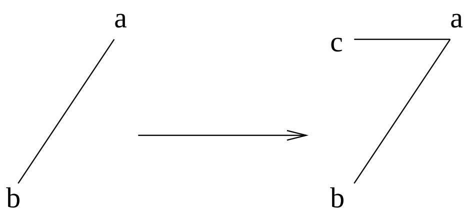

Fig. 16. Example of a specification graph.

11.2. *Reduction to the Usual Case*

11.2.1. *Rotation of the specification graph* We wish here to construct part of the code in charge of ensuring that we are left with solving a specification problem Δ such that $(u,\perp)\Delta(u,\perp)$ and $(\perp,v)\Delta(\perp,v)$.

Suppose $(u,\perp)\Delta(f(u),\perp)$ and $(\perp,v)\Delta(\perp,g(v))$. By Church's thesis, f and g are partial recursive functions. Then the program $Prog = P(f) \mid P'(g)$ with $P(f)$ and $P'(g)$ defined below solves the specification Δ if and only if $P \mid P'$ solves the specification Δ' with $(f(u),\perp)\Delta'(f(u),\perp)$, $(\perp,g(v))\Delta'(\perp,g(v))$ and $(f(u),g(v))\Delta'(f(u'),g(v'))$ whenever $(u,v)\Delta(u',v')$.

$$\begin{array}{rclcrcl} P(f) & = & (u = f(u)); & \quad & P'(g) & = & (v = f(v)); \\ & & P & & & & P' \end{array}$$

Proof. Using the standard semantics, we can show that the line of code before the calls to P and P' only acts on the local memory of each processor, hence there is no other action than the one deduced from the purely sequential behaviour of $P(f)$ and $P'(g)$ respectively. $\square$

11.2.2. *Minimal unfolding of the output graph* We now suppose that we have to solve a specification problem with a relation which is such that it is the identity relation when restricted to the vertices of the graph. We fulfill now the hypotheses of Theorem 2.

Let $e = ((P,u),(P',v))$ be any segment of the input graph, and G_e be the subgraph of the output graph (connected by Theorem 2), image of e by the specification relation Δ. Let $\overline{G_e}$ be the directed graph generated by G_e where each segment has an inverse. To exemplify the whole process described in this section, look at Figure 16 for the specification graph corresponding to a segment $e = (a,b)$ (the graph G_e is at the right-hand side of the picture), and to the left of Figure 17 for a picture of $\overline{G_e}$. An unfolding of G_e is any path p from $(u,\perp)$ to $(\perp,v)$ in $\overline{G_e}$ such that p traverses all segments of G_e. The minimal unfolding is the shortest of such paths. Its interest lies in the fact that from there we will be able to generate a code for P and P' that will implement this subpart of the specification graph. We will see in next section and in Section 11.4.2 that the length of this code is linearly related to the length of this unfolding, hence the usefulness of finding the shortest path to get the most efficient code.

An algorithm for determining such a minimal unfolding is based on a *breadth-first* traversing strategy (Sed88) of the graph, the traversing being complete when the criterion "having gone through all non-oriented segments and ending at $(\perp,v)$" is met. For

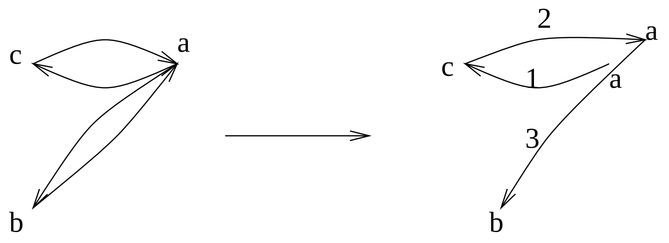

Fig. 17. Minimal unfolding (right) of the graph (left).

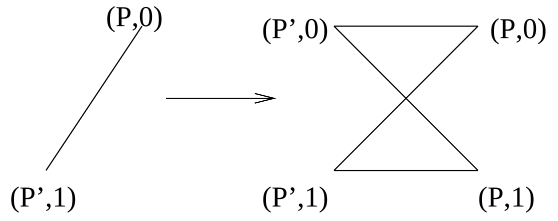

Fig. 18. A specification graph.

instance, this algorithm constructs the minimal unfolding of G_e which is pictured at the right of Figure 17.

Example 2.

— We can carry on the example specified in Figure 16, setting for instance $a = (P, x)$, $b = (P', y')$ and $c = (P', y)$ the program implementing the specification (i.e. the subdivision of the segment (a, b) into the minimal unfolding $((a, c), (c, a), (a, b))$) is $Prog = P \mid P'$ with,

$$
\begin{array}{rcl@{\qquad\qquad}rcl}
P & = & update; & P' & = & update; \\
 & & scan; & & & scan; \\
 & & case\ (u,v)\ of & & & case\ (u,v)\ of \\
 & & \ (x,y') :\ u = x; update & & & \ (x,y') :\ v = y; update \\
 & & \ default :\ update & & & \ default :\ update
\end{array}
$$

— Consider the specification graph pictured in Figure 18. The minimal unfolding is shown in two different ways in Figure 19. Using the result above, the code for implementing it is $Prog = P \mid P'$ with $P = P(0,0,0,0)[P(0,0,1,0)[\ P(1,1,1,0)]]$ and $P' = P'(0,0,0,0)[P'(0,0,1,0)[P'(1,1,1,0)]]$.

11.3. *An algorithm*

The specification graph is given. The algorithm terminates with an error (if the relation specified is not wait-free) or with the text of the two processes that implements the relation. The algorithm is as follows,

— Determine the rotation code (Section 11.2.1),
— For all segments $e = ((P, u), (P', v))$ of the input graph, do,

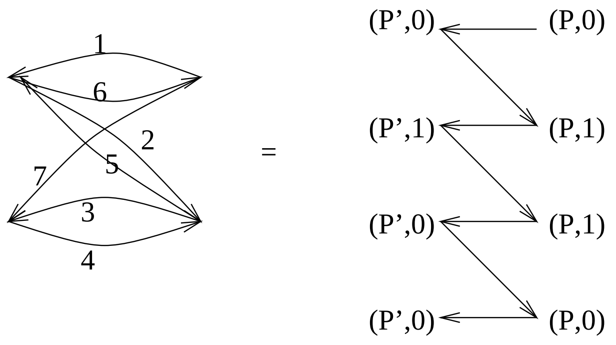

Fig. 19. The corresponding minimal unfolding and minimal path.

- determine the connected subgraph G_e of the output graph, image of e under the specification relation Δ,
- determine the minimal unfolding $((P,x_1)\dots(P,x_n),(P',y_n))$ of G_e (Section 11.2.2),
- The program up to that point is

$$Prog_e = P(x_1,\dots,y_n) \mid P'(x_1,\dots,y_n)$$

of Section 11.1.2,

— Mix the code for all segments.

We saw all the material needed in the previous sections except the "mixing" of the code for all segments. As a matter of fact, we have shown how to derive a code for the specification of just one input (a segment). Now we have to mix the codes for all inputs.

The idea here is quite simple: $Mix(Prog_1, Prog_2)$ ($Prog_1 = P_1 \mid P'_1$, $Prog_2 = P_2 \mid P'_2$) is essentially a program whose processes are $Mix(P_1,P_2)$ and $Mix(P'_1,P'_2)$ such that all their case entries are the union of the case entries of P_1 and P_2 (respectively of P'_1 and P'_2). Formally, Mix is an operation on processes that can be defined inductively when applied to the processes that subdivide segments

if $(x,y') \neq (X,Y')$,

```
$Mix(P(x,y,x',y')[P_1], P(X,Y,X',Y')[P_2]) =$
      $update;$
      $scan;$
      $case\ (u,v)\ of$
         $(x,y') : u = x'; update; P_1$
         $(X,Y') : u = X'; update; P_2$
         $default : update$
```

11.4. *Comparison with related work*

11.4.1. *The participating set and Herlihy's algorithm*

The participating set algorithm aims at solving the simplex agreement task of (Her94), that is, a generalization to any number of processors of the specification graph of Figure 14.

The intuition behind this algorithm is to subdivide all segments of the input graph, in

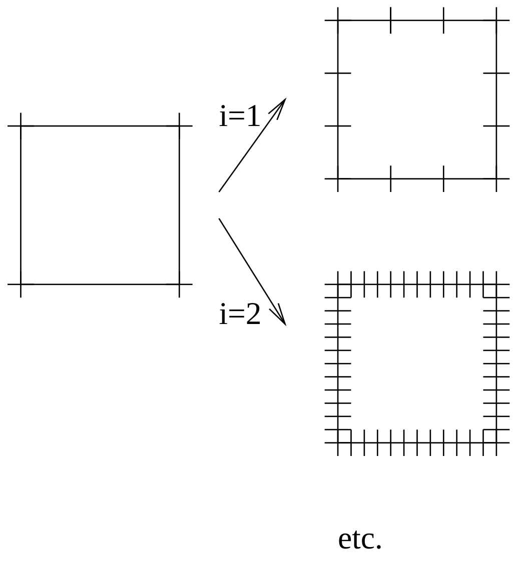

Fig. 20. Herlihy's iterated subdivision on the binary sphere.

a uniform manner, and enough so that all the subdivisions of the segments we need to implement the relation can be deduced from it. As a matter of fact, if we have subdivided a segment into N segments, then all subdivisions into M segments, $M \leq N$ can be deduced from it by just identifying the points in the finer subdivision which are not needed. The effect of the iterated participating set algorithm is (as shown in Figure 20) to create at iteration i a subdivision of all segments into 3^i segments.

11.4.2. *Complexity matters* As one might have already noticed, we have a strong relationship between the length of the minimal unfoldings, the number of times the program has to test the values of its variables, and the number of reads in the main memory. Let $t(e)$ be the maximum number of tests that $Prog$ has to make for all executions starting at segment e. Let $s(e)$ be the maximum number of $scan$ that $Prog$ has to execute for all executions starting at segment e. Then, calling $p(e)$ the minimal unfolding of G_e,

Lemma 9.

$$s(e) = t(e) = \frac{length(p(e)) - 1}{2}$$

Proof. Looking at the algorithm of Section 11.3, we see that all paths are recursively decomposed using the programs of type $P(x,y,x',y')[] \mid P'(x,y,x',y')[]$ such that at iteration z, we have subdivided e into a path of length $1 + 2z$. The cost in terms of tests and accesses to the main memory of each iteration is one. This entails the result. □

Whereas in case of Herlihy's algorithm we have up to $3 * max_e(s(e))$ accesses to the shared memory. In the case when all segments are mapped onto a segment except for one (like the one of Figure 21), the cost of computation is the same for all inputs and can be quite enormous.

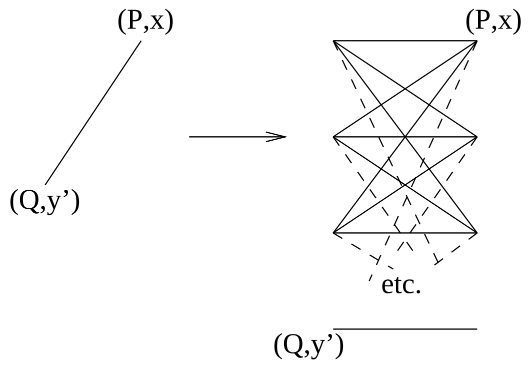

Fig. 21. The worst complexity case for a specification graph.

The algorithm proposed in this article is optimal in the sense that it minimizes $s(e)$ and $t(e)$ for all e whereas Herlihy's one subdivides all segments a power of three times uniformly.

Notice that the maximal complexity of the computation of wait-free relations on $[0, M] \cap \mathbb{Z}$ is not very high and is attained by our implementation for the specification graph shown in Figure 21 (for all input segments). It is such that for all inputs e, $s(e) = t(e)$ is asymptotically αM^2 with $\frac{1}{2} \leq \alpha \leq 1$.

Proof. In all G_e there are M^2 segments. Hence an unfolding of $\overline{G}_e$ has at least M^2 segments and at most $2M^2$ segments. We use Lemma 9 to conclude. □

12. A General Methodology

What we have done in dimension two is a good example of what we could try to do on more general architectures.

First of all, we have to explain the unfolding in the control flow semantics. To do this, let us represent the control flow in a slightly different manner, reminiscent of what has been done on process graphs (CRJ87; Dij68; Gun94). Let us represent commutation of two 1-transitions by filling their interleaving by a 2-transition, and represent non-commutation by not filling the interleaving with 2-transitions. This amounts to identifying the control flow of our asynchronous language with a semaphore program, for which the pair (*scan*, *update*) of actions is identified with an exchange of information, by P/V synchronisation. This is a good analogue up to the extent we are only interested by the effect of the history of the communications on the environment, look at Figure 22.

We then have mainly two configurations of "holes" on a square, when we look at the homotopy classes of directed paths (1-schedules), as shown in Figure 23[§].

In the first configuration, there are three 1-schedules (when the holes are "incomparable"), whereas in the second configuration, there are four 1-schedules (when the holes are "comparable"). Therefore, if you look at some more complex configuration between

[§] There can be no overlapping of holes as in the case of P/V programs

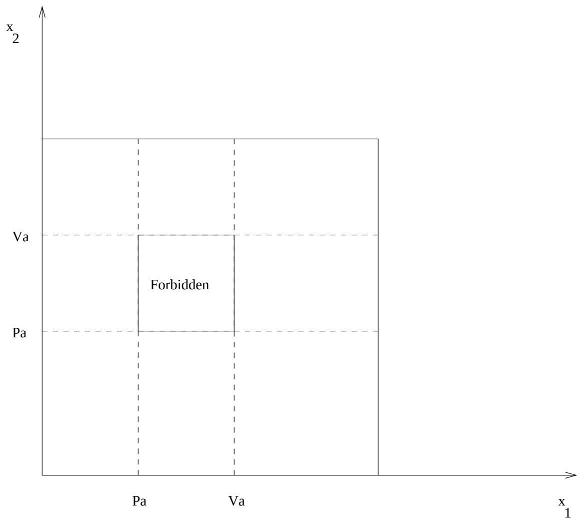

Fig. 22. A P/V analogue to *scan/update*

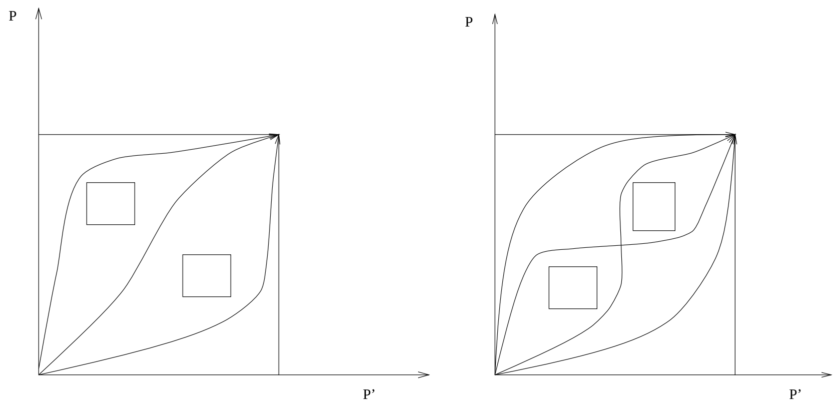

Fig. 23. The two possible relative configurations of holes

many holes, as in Figure 24, its set of 1-schedules is described by a complex tree-like picture (also in Figure 24).

As a matter of fact, we could easily describe a superset of the 1-schedules. To do this, look at Figure 25.

This was basically what we have been doing in the control flow semantics, by unfolding or tearing the shape of Figure 25.

Formally, two holes are comparable if there is a directed path from the end of one of the holes to the start of the other hole. Comparability is a partial order, and we can show that any linearisation of this partial order (one of which is pictured in Figure 25, by deforming the shape to have the holes in a linear configuration) gives a superset of the possible 1-schedules. But a chain (under this order, compatible with the comparability

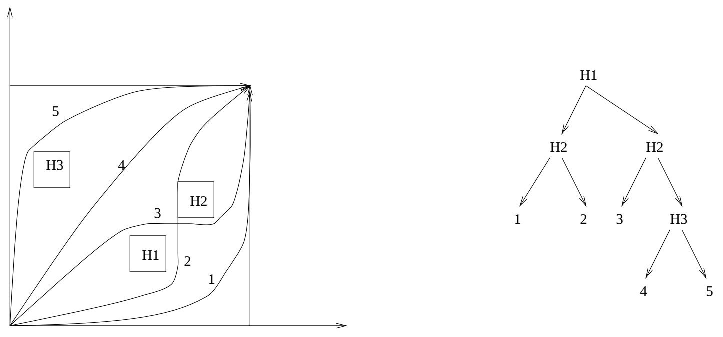

Fig. 24. A more complex situation

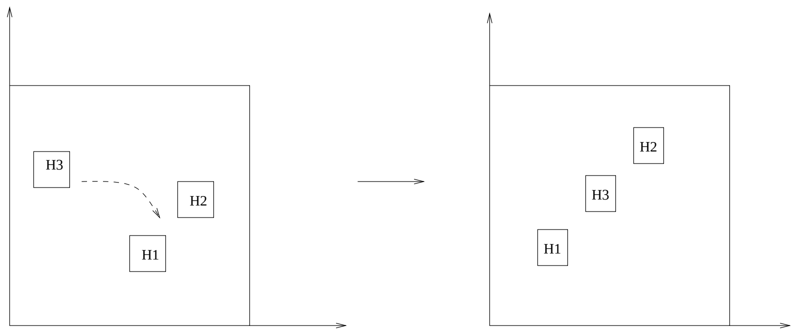

Fig. 25. A method to compute a superset of 1-schedules

partial order) of holes has exactly the "directed" homotopy type of a binary tree (Figure 26).

To any leaf of the tree is associated a 1-simplex in the output graph. In the semantics of the *scan/update* language, going from a leaf of the binary tree from the next one, we always share one vertex (a P vertex or a P' vertex), so the complex that is reached by the possible 1-schedules is connected. We have made this sketch of proof for a superset of

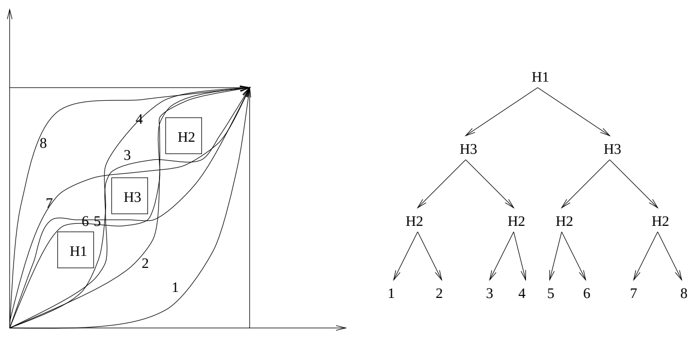

Fig. 26. The homotopy type of a chain of holes

possible 1-schedules. This does not change the connectivity argument, which completes the proof of Theorem 2.

This gives us a precise idea of how to study the expressiveness of distributed languages or architectures.

First, define the semantics of your system in such a way that you can formalise n-schedules (for instance using HDA, or using full information complexes as in (Her94)). To be more precise, you should specify the control flow part and the information flow part of the semantics. There is a great deal of flexibility when it comes to defining the "control flow" semantics. In this section we choosed to use some kind of P/V analogue, whereas in the previous sections, we abstracted the control flow directly from the semantics. Basically, we need only to be able to define the n-schedules so any weakly homotopy-equivalent (in the "directed" sense) spaces would do.

Then you might be able to prove, as a side effect, that your system is t-resilient (i.e. the dynamics is t-connected). All k-schedules (or homotopy classes of directed k-paths) in the control flow semantics may give a $(k-1)$-simplex in the output complex, that represent a possible final global state of your system. The best is to characterise a superset of these k-schedules, since it is in general very difficult to fully characterise the precise set of k-schedules itself.

Then you might be able to find a natural order on these k-schedules, because in general the control flow semantics will have the "directed" homotopy type of some kind of tree structure. Going from one schedule to the next one (under that order) means modifying in the most elementary manner the histories, hence the knowledge that one processor has from the other at some time of computation. This part is given to you by the semantics of the medium through which interaction or communication occurs. Using the information flow semantics, you might show that some boundaries of these $(k-1)$-simplexes associated with these k-schedules are common, since the change in the information is not so radical when going from one schedule to the next one. This methodology will be applied briefly in the two next sections. The first one deals with higher-dimensional schedules. The second one shows the difference of result we might expect if we do not change the control flow dynamics, but just change the information flow semantics.

13. Some Hints in Higher-Dimensions

The machine we are considering now has n processors $P_1, \cdots, P_n$ communicating through a shared memory, in a similar manner as in Section 4. Processor P_i can only write on the location x_i, by an *update* of its local value u_i^i. It can also *scan* the whole memory and stores all x_j in its local registers u_i^j.

The semantics will basically follow the same rules that we had taken for the 2-dimensional case. The skeleton of dimension one of $[\![P]\!]\rho$ ($P = (P_1 \mid \cdots \mid P_n)$) is the one defined by the standard concrete semantics of Section 4 where the states are now of the form $(\{P_1, \cdots, P_n\}, \rho)$. Now we have the following lemma, generalizing Lemma 1.

Lemma 10. (look at Figure 27) Let

$$a_i \quad = (\{t_1; P_1, \cdots, t_n; P_n\}, \rho_\epsilon) \xrightarrow{t_i} (\{t_1; P_1, \cdots, P_i, \cdots, t_n; P_n\}, \rho_i)$$

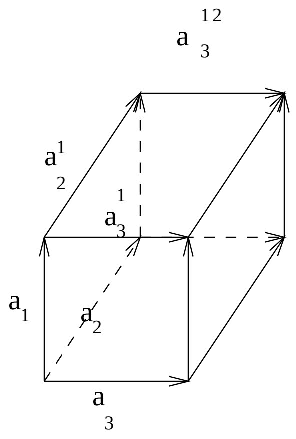

Fig. 27. n non-interfering transitions

be a 1-transition for all $1 \leq i \leq n$. We have the following 1-transitions that can be fired from the end states of these 1-transitions, for all $S \in (\{1, \cdots, n\})^*$ (a string on the alphabet $\{1, \cdots, n\}$), if $i \notin S$ (meaning i does not appear in the string$S$),

$$a_i^S \ =(\{\cdots, P_j, \cdots, t_k; P_k, \cdots\}, \rho_S) \xrightarrow{t_i} (\{\cdots, P_j, \cdots, P_i, \cdots, t_k; P_k, \cdots\}, \rho_{S.i})$$

where the index j ranges over the indices that do appear in S and the index k ranges over the indices that do not appear in S, except index i. Then if no (t_i, t_j) is equal to $(scan, update)$ then

$$\rho_{1\cdots k} = \rho_{\sigma(1)\cdots\sigma(k)}$$

for all k and for all permutations σ on $\{1, \cdots, k\}$.

Proof. By induction on n and case analysis on the t_i, using the SOS semantics of Section 4. This only states the fact that all actions but *scan* and *update* commute in the standard concrete semantics. □

In the HDA framework, we decide to make all these actions (the ones of Lemma 10) completely asynchronous, therefore we should have a n-transition within these 1-transitions. This is denoted as follows.

The n-transition generated by the n 1-transitions $a_1, \cdots, a_n$ is denoted by $a_1 \otimes \cdots \otimes a_n$. A n-transition has n $(n-1)$-dimensional start boundaries and n $(n-1)$-dimensional end boundaries. These will be specified as in the 2-dimensional case: a n-transition will go from a list of n $(n-1)$-transitions to a list of n $(n-1)$-transitions.

In the case of Lemma 10, we have the following rules,

$(non - interference)$

$$\otimes_{i\neq 1} a_i, \cdots, \otimes_{i \neq n} a_i \xrightarrow{a_1 \otimes \cdots \otimes a_n} \otimes_{i\neq 1} a_i^1, \cdots, \otimes_{i\neq n} a_i^n$$

Now the possible interferences in the n-dimensional case are combinatorially more complex than in dimension 2. The principle, though, is very similar.

Suppose that we have two subsets $S, U \subseteq \{1, \cdots, n\}$ with $S \cap U = \emptyset$, $S \cup U = \{1, \cdots, n\}$

such that S is the set of indices i for which $t_i = scan$ and U is the set of indices i for which $t_i = update$. Let us call (U,S)-shuffle any sequence of sets $T = (T_i)_{1 \leq i \leq 2k}$ such that,

— each T_i is a (possibly non-proper) subset of S or of U,
— $T_0 = \{T_{2i}/1 \leq i \leq k\}$ forms a partition of S,
— $T_1 = \{T_{2i-1}/1 \leq i \leq k\}$ forms a partition of U,
— for all $i < k$, $T_{2i} \neq \emptyset$,
— for all $i > 1$, $T_{2i-1} \neq \emptyset$.

We have a weaker lemma than Lemma 10, but we can still recognize that some of the states might just be the same. If s is a string on the alphabet $\{1, \cdots, n\}$, we call $l(s)$ the set of letters that s is made of.

Lemma 11. Let s be a string (with no repetition) on the alphabet $\{1, \cdots, n\}$. Given S and U two disjoint subsets of $\{1, \cdots, n\}$ and T a (U,S)-shuffle, we say that $s \in T$ is and only if s can be decomposed as $s = s_1 s_2 \cdots s_{2k}$ with,

$$l(s_i) = T_i$$

Then whenever s and s' are two strings in the same shuffle T, $\rho_s = \rho_{s'}$.

Proof. Tedious case analysis proof on the semantics. Basically, any path that has gone through *updates* of some fixed set of processes and then through *scans* of some other fixed set of processes etc. should end up in a state depending only on these sets of processes. □

Now we generate one n-transition for each (U,S)-shuffle, and unfold this to separate out all future executions of all different (U,S)-shuffles, as we have be doing in the 2-dimensional case¶.

Basically, all these n-transitions will have as start $(n-1)$-transitions the transitions $\otimes_{i \neq 1} a_i, \cdots, \otimes_{i \neq n} a_i$ as in the non-interfering case. All these n-transitions will lead to the environment characterized by a given (U,S)-shuffle, and include all of the 1-skeleton (defined by the standard SOS semantics). This leads to n-dimensional shapes like the one pictured in Figure 28, the higher-dimensional counterpart of the paper clips we saw in dimension 2.

Let us call C_T the n-transition generated by the (U,S)-shuffle T. Then, as there are less that $n!$ of these shuffles, we can give a ordering on these such that the levels are dealt with using the following rules, if $0 \leq lv(T) \leq n!$ and

$$A_1, \cdots, A_n \xrightarrow{C_T} A'_1, \cdots A'_n$$

then all states of $A'_1, \cdots, A'_n$ (which are not in $A_1, \cdots, A_n$, whose states were of level l) are of level $n!l + lv(T)$ (this is an unfolding of the n-schedules).

The 1-skeleton of the semantics remains basically the same as we had in the 2-processor case, so the proofs of the following facts remain essentially the same,

¶ There was only two possible shuffles in dimension 2, namely one corresponding to doing *scan* before *update*, and the other corresponding to doing *update* before *scan*.

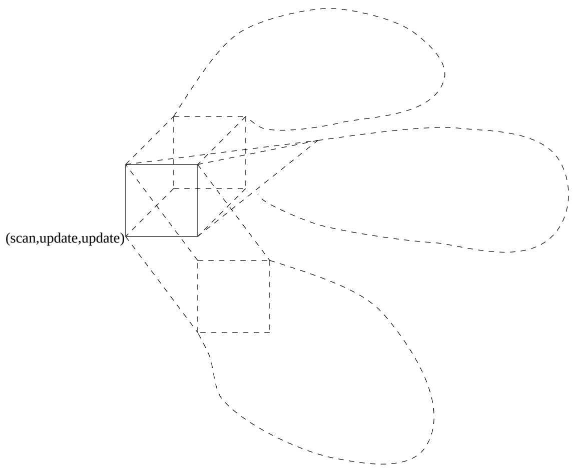

Fig. 28. A 3 dimensional paper clip

— The HDA semantics of any term of the language is connected and acyclic,
— There is always a unique solo execution (of a given process) from a given global state,
— The control flow semantics of any term is wait-free.

Each cut in the final cut semantics is a n-simplex. The specification graph is now a specification complex, which is a relation between an input complex and an output complex, generalising the input, output graphs of Section 9.

There again, a critical point c is any state for which the rule (*interference*) can apply. If k is the cardinal of the set S of indices of *scan* operations in c and l is the cardinal of the set U of *update* operations, there are, say $s(c,l)$ shuffles and the critical point ("of index s") creates s different n-dimensional layers. Again each of these layers is in bijection both with the set of final environments and the final cuts. These cuts are $(n-1)$-simplices. If the layers are numbered from 1 to m, the corresponding $(n-1)$-simplices are $(\alpha_1^1, \cdots, \alpha_n^1), \cdots, (\alpha_1^m, \cdots, \alpha_n^m)$. We prove that going from layer i to layer $i+1$ we have at most an index j such that $\forall r \neq j, \alpha_r^{i+1} = \alpha_r^i$. First of all we notice that this condition is actually a very geometric one,

Lemma 12. Let $(\alpha_1^1, \cdots, \alpha_n^1), \cdots, (\alpha_1^m, \cdots, \alpha_n^m)$ be a set of m $(n-1)$-simplices such that $\forall i, \exists j, \forall r \neq j, \alpha_r^{i+1} = \alpha_r^i$. Then the union of these $(n-1)$-simplices is a connected, $(n-2)$-connected shape.

Proof. We prove this by induction on m. When $m = 1$ this is obvious since we have only one $(n-1)$-simplex which is connected, $(n-1)$-connected hence also connected, $(n-2)$-connected.
When $m > 1$, we have to glue together the first $m-1$ $(n-1)$-simplices (space A) with the last $(n-1)$-simplex (space B). A is a connected, $(n-2)$-connected shape by the induction hypothesis. We have also seen that B is connected and $(n-2)$-connected. The condition $\forall i, \exists j, \forall r \neq j, \alpha_r^m = \alpha_r^{m-1}$ means that $A \cap B$ is a $(n-2)$-simplex or a $(n-1)$-simplex (if $\alpha_j^m = \alpha_j^{m-1}$ holds as well, for instance), thus is at least a connected, $(n-2)$-connected

shape. Applying Lemma 5 proves that $A \cup B$ is connected, $(n-2)$-connected, i.e. the result for m. $\square$

To prove that the hypothesis of this lemma holds, we need to enumerate the different levels (or layers).

Let U and S be fixed subsets of $\{1, \cdots, n\}$. Let T be a (U,S)-shuffle. For all $s \in S$ there is a unique i in $\{1, \cdots, k\}$ such that $s \in T_{2i}$. Consider now $\cup_{1 \leq j \leq i} T_{2j-1}$. It is a set of indices in $\{1, \cdots, n\}$. We can represent it as a suborder of $\{1, \cdots, n\}$, or a string $up_T(s)$.

Let now T and T' be two (U,S)-shuffles. S being the set $s_1 < \cdots < s_k$, we say that $T \leq_{(U,S)} T'$ if and only if the string $up_T(s_1) \cdots up_T(s_k)$ is less or equal than $up_{T'}(s_1) \cdots up_{T'}(s_k)$ in the lexicographic ordering. Then,

For all U and S disjoint subsets of $\{1, \cdots, n\}$, $\leq_{(U,S)}$ is a total ordering on the set of (U,S)-shuffles.

Now at a critical point of index s (depending on sets U and S) the different values environment can take when going from a layer to one of the next ones are as follows,

— change the value of P_{s_1},
— or change the value of P_{s_2},
— $\cdots$,
— or change the value of P_{s_k}.

So the hypothesis of Lemma 12 is satisfied and the image of any k-simplex under the denotational relation is a $(k-1)$-connected complex.

14. Some Hints about Wait-free test&set Protocols

In this section we add to the language a test&set operation ($t\&s$) on a flag f shared by the processes $P_1, \cdots, P_n$, as a case example of the methodology outlined in Section 12 . This is done by extending the case statement to include a test on $t\&s(f)$. This simple extension to the language changes quite dramatically what kind of relation it can compute.

The semantic rule for (*case*) should be changed as follows,

(*case*)

If $\exists k$, $\forall i$, $u_i = a_i^k$ and $f = \alpha_k$,

$$\left(\left\{\begin{pmatrix} case\ (u_1 \ldots u_k, f)\ of \\ (a_1^1 \ldots a_k^1, \alpha_1) :\ P_1 \\ \cdots \\ (a_1^n \ldots a_k^n, \alpha_n) :\ P_n \\ default :\ P \end{pmatrix} ; R, P' \right\}, \rho\right) \xrightarrow{case_P} (\{P_k; R, P'\}, \rho)$$

Otherwise,

$$\left(\left\{\begin{pmatrix} case\ (u_1 \ldots u_k, f)\ of \\ (a_1^1 \ldots a_k^1, \alpha_1) :\ P_1 \\ \cdots \\ (a_1^n \ldots a_k^n, \alpha_n) :\ P_n \\ default :\ P \end{pmatrix} ; R, P' \right\}, \rho\right) \xrightarrow{case_P} (\{P; R, P'\}, \rho)$$

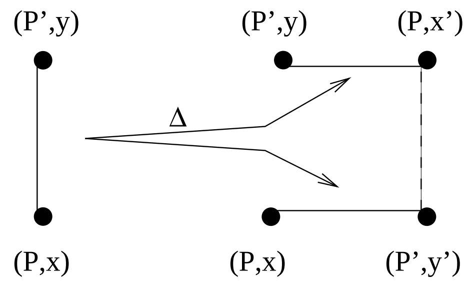

Fig. 29. The splitting of two segments in a specification graph

Let us first look at the case $n = 2$.

Lemma 13. The specification graph of Figure 29 can be implemented in our new language.

Proof. The following program implements the "splitting" of one segment into two others,

```
Prog = P | P'
P    = update;                              P' = update;
       scan;                                     scan;
       case (u, v, t&s(f)) of                    case (u', v', t&s(f)) of
          (x, ⊥, 0) : u = x; update                 (⊥, x', 0) : v' = y; update
          (x, z, 1) : u = y; update                 (x, z, 1) : v' = t; update
```

The value of $t\&s(f)$ is found equal to 0 by the first process which tests it, and is found equal to 1 by the second process which tests it. □

In particular, the binary consensus can be solved using test&set. The dynamics of the language with test&set is just the same as the dynamics of the language without. The only difference is that the flag enables any process to know if one or more processes have gone through a case statement (and thus executed a test&set operation), so that any process can have a (very) partial view of the past history of the interactions (through the *scan*s and *update*s).

In dimension 2, when going from layer i to layer $i+1$ there is no reason why we should have $\alpha_{i+1} = \alpha_i$ or $\beta_{i+1} = \beta_i$.

A reciprocal holds in a quite straightforward way,

Theorem 3. Any specification graph such that the image if a segment (u, v) under the specification relation is a union of a finite number of connected components, containing moreover points $(u, \perp)$ and $(\perp, v)$ can be implemented in a wait-free manner in our test&set language.

Proof. The algorithm is quite straightforward. First split segments as much a you need to by using the program in the proof of Lemma 13. Then use the subdivision algorithm of Section 11.3. □

In dimension n in general, we conjecture that we have the following phenomenon. There might only exist two indices j and j' such that $\forall r \neq j, j' \alpha_r^{i+1} = \alpha_r^i$. This condition comes from the fact that there might be a partial knowledge of the history of communications (by using test&set) only by no more than two processors at a time, therefore, only two processors could change their decision values when going from a layer to the next one.

We notice that this condition, as in the case of the language without $t\&s$, is actually a very geometric one,

Lemma 14. Let $(\alpha_1^1, \cdots, \alpha_n^1), \cdots, (\alpha_1^m, \cdots, \alpha_n^m)$ be a set of m $(n-1)$-simplices such that $\forall i, \exists j, j', \forall r \neq j, j', \alpha_r^{i+1} = \alpha_r^i$. Then the union of these $(n-1)$-simplices is a connected, $(n-3)$-connected shape.

Proof. We prove this by induction on m. When $m = 1$ this is obvious since we have only one $(n-1)$-simplex which is connected, $(n-1)$-connected hence also connected, $(n-3)$-connected.
When $m > 1$, we have to glue together the first $m-1$ $(n-1)$-simplices (space A) with the last $(n-1)$-simplex (space B). A is a connected, $(n-3)$-connected shape by the induction hypothesis. We have also seen that B is connected and $(n-3)$-connected. The condition $\forall i, \exists j, j' \forall r \neq j, j' \alpha_r^m = \alpha_r^{m-1}$ means that $A \cap B$ is a $(n-3)$-simplex or a $(n-2)$-simplex or a $(n-1)$-simplex (if $\alpha_j^m = \alpha_j^{m-1}$ or $\alpha_{j'}^m = \alpha_{j'}^{m-1}$ holds as well, for instance), thus is at least a connected, $(n-3)$-connected shape. Applying Lemma 5 proves that $A \cup B$ is connected, $(n-3)$-connected, i.e. the result for m. $\square$

Therefore, we conjecture that our new machine has the following characterisation. The image of any k-simplex of the input graph under the specification relation is $(k-1)$-connected $(k \geq 1)$.

15. Conclusion

We have presented in this article some examples of semantic formalisations of the expressive power of some distributed machines and architectures. This is obviously the first step only, towards purely mathematical characterisations of distributed computing. For this purpose, we would need a complete formalisation of a theory of "directed" homotopy, as advocated in (Gun94). We unfortunately have only very few results in general, and we cannot always resort to homological characterisations as we have done here, or as done in (Goult) (where some kind of "directed" homology was studied).

One of the open problems in that respect would be to fully characterize t-resilient computations (using a language with some semaphores initialized to $n-t-1$, or some synchronisation barriers involving no more than $n-t-1$ processors) and see if we can even find a nice normal form to programs, as for shared memory wait-free computations.

References

E. Borowsky and E. Gafni. Generalized FLP impossibility result for t-resilient asynchronous computations. In *Proc. of the 25th STOC.* ACM Press, 1993.

E. Borowsky. Capturing the power of resiliency and set consensus in distributed systems. Technical report, University of California in Los Angeles, 1995.

P. Cousot and R. Cousot. Inductive definitions, semantics and abstract interpretation. In *Conference Record of the 19th ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Programming Languages*, pages 83–94. ACM Press, 1992.

S. Chaudhuri. Agreement is harder than consensus: set consensus problems in totally asynchronous systems. In *Proc. of the 9th Annual ACM Symposium on Principles of Distributed Computing*, pages 311–334. ACM Press, August 1990.

S. D. Carson and P. F. Reynolds Jr. The geometry of semaphore programs. *ACM Transactions on Programming Languages and Systems*, 9(1):25–53, January 1987.

E.W. Dijkstra. *Cooperating Sequential Processes.* Academic Press, 1968.

M. Fisher, N. A. Lynch, and M. S. Paterson. Impossibility of distributed commit with one faulty process. *Journal of the ACM*, 32(2):374–382, April 1985.

E. Goubault and T. P. Jensen. Homology of higher-dimensional automata. In *Proc. of CONCUR'92*, Stonybrook, New York, August 1992. Springer-Verlag.

E. Goubault. Domains of higher-dimensional automata. In *Proc. of CONCUR'93*, Hildesheim, August 1993. Springer-Verlag.

E. Goubault. Schedulers as abstract interpretations of HDA. In *Proc. of PEPM'95*, La Jolla, June 1995. ACM Press, also available at http://www.ens.fr/~goubault.

E. Goubault. A semantic view on distributed computability and complexity. In *Proceedings of the 3rd Theory and Formal Methods Section Workshop.* Imperial College Press, also available at http://www.ens.fr/~goubault, 1996.

E. Goubault. *The Geometry of Concurrency.* PhD thesis, Ecole Normale Supérieure, to be published, 1995, also available at http://www.ens.fr/~goubault.

J. Gunawardena. Homotopy and concurrency. In *Bulletin of the EATCS*, number 54, pages 184–193, October 1994.

P. Gabriel and M. Zisman. Calculus of fractions and homotopy theory. In *Ergebnisse der Mathematik und ihrer Grenzgebiete*, volume 35. Springer Verlag, 1967.

M. Herlihy. A tutorial on algebraic topology and distributed computation. Technical report, presented at UCLA, 1994.

M. Herlihy and S. Rajsbaum. Set consensus using arbitrary objects. In *Proc. of the 13th Annual ACM Symposium on Principles of Distributed Computing.* ACM Press, August 1994.

M. Herlihy and S. Rajsbaum. Algebraic topology and distributed computing, a primer. Technical report, Brown University, 1995.

M. Herlihy and N. Shavit. The asynchronous computability theorem for t-resilient tasks. In *Proc. of the 25th STOC.* ACM Press, 1993.

M. Herlihy and N. Shavit. A simple constructive computability theorem for wait-free computation. In *Proceedings of STOC'94.* ACM Press, 1994.

N. Lynch. *Distributed Algorithms.* Morgan-Kaufmann, 1996.

J. P. May. *Simplicial objects in algebraic topology.* D. van Nostrand Company, inc, 1967.

S. Mac Lane. Homology. In *Die Grundlehren der Mathematischen Wissenschaften in Einzeldarstellungen*, volume 114. Springer Verlag, 1963.

V. Pratt. Modeling concurrency with geometry. In *Proc. of the 18th ACM Symposium on Principles of Programming Languages.* ACM Press, 1991.

B. Sedgewick. *Algorithms.* Addison-Wesley, 1988.

M. Saks and F. Zaharoglou. Wait-free k-set agreement is impossible: The topology of public knowledge. In *Proc. of the 25th STOC.* ACM Press, 1993.

R. van Glabbeek. Bisimulation semantics for higher dimensional automata. Technical report, Stanford University, Manuscript available on the web as http://theory.stanford.edu/~rvg/hda, 1991.