A SEMANTIC VIEW ON DISTRIBUTED COMPUTABILITY AND COMPLEXITY

E. GOUBAULT
CNRS & Ecole Normale Supérieure
45 rue d'Ulm, 75005 PARIS, FRANCE

Abstract

This paper intends to give a semantical perspective on the recent work by Herlihy, Shavit and Rajsbaum on computability and complexity results for t-resilient and wait-free protocols for distributed systems. It is an extended abstract[a] of a talk given at the Imperial College Workshop, Oxford, Christ Church on the 2nd of April 1996.

1 Introduction

In this article we address some computability and complexity problems which have most often arisen in the area of protocols for distributed systems and concurrent databases. The essence of these problems is to decide whether we can compute a certain kind of *function* in a distributed - yet *robust* - manner. Let us take our first example from the concurrent database theory. Imagine that we have a database that can be shared by n concurrent transactions $T_1, \cdots, T_n$ asynchronously. We suppose that the network linking the transactions to the shared database is not reliable in the sense that any wire can be cut unexpectedly, and no transaction can test the failure of a wire - or equivalently, the processors supporting each transaction may fail without prior notice to any of the others. So we would like our transactions to be managed by a protocol, that is by a program on each processor, which can handle all sorts of failure and still ensure some basic properties on the transactions which have not failed. This means that we want the transactions to be as loosely coupled as possible so that a failure of one processor will not prevent others that might wait for an answer from deadlocking. What functions can these transactions compute then? Let us make precise what sort of function we are interested in. Suppose all transactions T_i want to change the value of the same item x to (maybe) different values x_i. Can all transactions agree on a common value, taken from the set $x_1, \cdots, x_n$? A classic result is that as soon as we ask for a certain robustness to failure, this cannot be computed on ordinary atomic read/write register machines[4]. We will derive a new proof of this result and other similar

[a]Many thanks to Sergio Rajsbaum and Patrick Cousot for useful discussions. I have used Paul Taylor's macro package for drawing diagrams.

ones in an entirely semantic framework, giving another perspective on recent work[1,2,10,11,12,13,14,17].

In section 2 we define formally the class of *functions* we are interested in computing, then define the class of distributed *machines* that we are considering for the computation. We relate this class to the geometric considerations of E. Goubault[6,8] and in Section 3 and 4 we derive the main decidability (or computability) results, giving examples in a toy language. In Section 4 we generalize these questions in order to define the complexity of the computation of functions allowed on some machines.

2 The framework

2.1 The functions

We will actually try to compute *relations* and not just functions. To account for the distributed nature of their computation, we will consider that the input arguments and output values of the relations are tuples $(x_1, \cdots, x_n)$ of integers. The idea is that each x_i is the local view of the input (or of the output) on processor i, on a system which might use at most n processors (that is at most n independent threads of execution). So the class of relations $\mathcal{R}_n$ we are interested in is just the set of binary relations on $\mathbb{Z}^n$. Let us give a few examples. The relation considered in the introduction is the relation *consensus* Γ defined by $(x_1, \cdots, x_n)\Gamma(x_i, \cdots, x_i)$ for any tuple $(x_1, \cdots, x_n) \in \mathbb{Z}^n$ and $i \in \{1, \cdots, n\}$. This means that the n processors can elect any of them, and choose thier local value as the common local view of the result. The *binary consensus* relation Γ_b is defined in a similar manner, except its domain is restricted to $\{0, 1\}^n \subseteq \mathbb{Z}^n$. Suppose now that we equip $\mathbb{Z}^n$ with the lattice structure induced by the order $(x_1, \cdots, x_n) \leq (y_1, \cdots, y_n)$ if and only if $x_1 \leq y_1$ and$\cdots x_n \leq y_n$. The ordered consensus relation Δ represents the fact that the election takes the greatest input value as the result, $(x_1, \cdots, x_n)\Delta(x_i, \cdots, x_i)$ where $x_i = \vee_{1 \leq j \leq n} x_j$. Notice that the *ordered binary consensus* is nothing but the *parallel or* (on n booleans). Pseudo-consensus is a weak version of the consensus in the sense that we allow some errors in the agreement between the transactions: the first processor might agree on the the common value plus one (so that the error is at most one on the agreement). Formally, this is the relation Ψ such that, $(x_1, \cdots, x_n)\Psi(x_i, \cdots, x_i)$, $(x_1, \cdots, x_n)\Psi(x_i+1, x_i, \cdots, x_i)$

2.2 The machines

Suppose we have a machine with n processors. Then the machine (or a program on the machine) is t-resilient ($t \leq n - 1$) if it can terminate any computation even if t among its n processors fail. This means not only that the whole system will terminate in case of t failures but also that the "partial result" computed by the remaining $n-t$ processors will be a partial view of the result that one would have obtained if ever the n processors would have completed their executions. When $t = n-1$ we say that the machine (or the program) is wait-free. In some way, we are just asking for the processors to be as loosely coupled as possible so that no one has actually to wait for another. This obviously limits the amount of information one process can have from the others at any moment of the execution. When you look at the possible schedulings of actions, all permutations of actions are allowed in a wait-free program since there should be no waiting between processors. The amount of permutation allowed for t-resilient programs is less in general. This will motivate the approach using a characterization of the possible schedulers of the machines (Section 4).

These requirements on the machines will also slightly change the formulation we had on the relations we wish to compute (Section 2). If we use the most famous symbol $\perp$ to denote failure or non-termination, then all relations R we wish to compute on a t-resilient machine should actually be extended over $(\mathbb{Z} \cup \{\perp\})^n$ so that they satisfy, $(\cdots, x_{i-1}, \perp, x_{i+1}, \cdots) R (\cdots, y_{i-1}, \perp, y_{i+1}, \cdots)$ and $(x_1, \cdots, x_n) R (y_1, \cdots, \perp, y_{i_1}, \cdots, \perp, y_{i_2}, \cdots)$ where there could be at most t $\perp$ in the right-hand side of the last equation.

Example of a machine

An interesting case is a shared memory machine with atomic read/write registers[14]. The shared memory is formalized by a collection of registers x_i in a set V. Each processor P_i has got a local memory composed of registers r_j^i in a set V_i. 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_i is only allowed to write on x_i). We will use the following syntax for our machine; first we have a grammar for instructions I, and then one for processes P,

$$I := \mathbf{nil} \ | \ scan \ | \ write(u) \ | \ r = f(r_1, \cdots, r_n)$$

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

$$P := I \ | \ branch(r, I, I) \ | \ P; P$$

where r is any local register. Programs are $Prog := (P \mid Prog)$.

nil is the instruction that writes the local value of processor P_i (i.e. r_i^i) in the shared variable x_i. an action). It is used as one of the branches of the if statement (called *branch* here) if we do not need to do anything in the alternative case. *scan* reads the shared array in one round and stores it into some of the local registers of the process in which it is executed. If $x_1, \cdots, x_k$ is the shared array in the shared memory, and r^i_j is the jth local register of process P_i, then *scan* executed in P_i stores $x_1, \cdots, x_k$ in $r^i_1, \cdots, r^i_k$. We suppose (for convenience) that it also writes its local value (r^i_i for processor i) in the shared variable x_i. $write(u)$ executed in P_i writes u in x_i. $r = f(r_1, \cdots, r_n)$ computes the partial recursive function f with arguments $r_1, \cdots, r_n$ and stores the result in r. $branch(r, i_1, i_2)$ where i_1 and i_2 are two instructions, tests whether r is strictly positive or not (it is an "instant decision", see the semantics). If it is strictly positive then it executes i_1 otherwise it executes i_2. ; is the sequential composition of processes. $|$ is the parallel composition of processes.

Let us step back for a second, and look at a sample program in our language.

$$
\begin{array}{rcl}
Prog & = & P \mid Q \\
P & = & scan; \\
 & & branch(r_2, write(1), \mathbf{nil})
\end{array}
\qquad
\begin{array}{rcl}
 & & \\
\mathrm{Q} & = & scan; \\
 & & branch(r_1, \mathbf{nil}, write(0))
\end{array}
$$

This program achieves binary pseudo-consensus (look at the formal semantics of next section). Remark that there is indeed no waiting (i.e. synchronisation) between processes. Therefore, if a processor was shut down in the middle of the computation, the others could carry on their respective computations. More generally, we can see that in this language, even if t processors fail ($1 \leq t \leq n-1$), the remaining processes can carry on their computations. Now, what relations can we compute in that language?

The concrete semantics

We model the concurrent execution of sequential processes by using Higher-Dimensional Automata (HDA[16,5]). Let us first explain informally the geometric ideas underlying the formalization with HDA. If we had a local clock with continuous time on each processor[7], sequential observations of concurrent executions of our system could be viewed as 1-dimensional curves, or trajectories, parameterized by the local time of each processor taking part in the execution. In general we associate one coordinate in the Euclidean space R^n, say x_i to each processor p_i in the set of available processors $p_1, p_2, \cdots, p_n$. For instance, in (i) of Figure 1, two processors are firing actions a, b concurrently, each of them with a local time in $[0,1]$. So the possible trajectories will be paths inside the square $[0,1] \times [0,1]$. As we suppose that our processors "consume" time for computation, paths should also be strictly increasing in every coordinate. In our example, none of the processors interfere in any manner, so the sequential

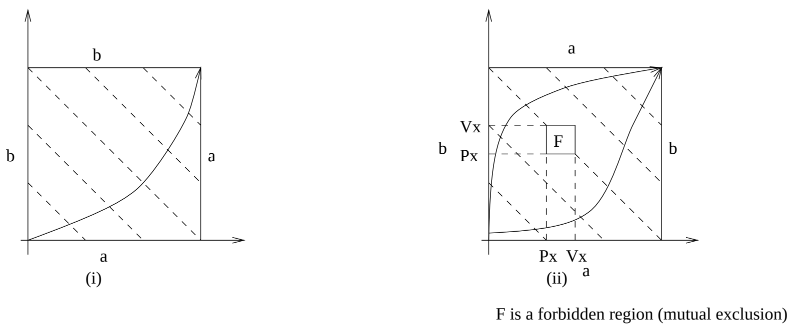

Figure 1: Difference between non-interfering (i) and interfering (ii) parallel composition.

observer might observe any of the trajectories within the square $[0, 1] \times [0, 1]$. Notice that we are not knowing the exact computations made by the actions of the two processors. Thus the fact that they are non-interfering, therefore the fact that any sequential *schedule* of the two actions give the same result (terminates in the same global state) is reflected by the *geometric* fact that all the paths from the initial to the final state (and in particular the two interleavings of the two actions) are *homotopic*. This means that one can always be continuously deformed through the surface of the square into the other. The idea is that any action a is actually made up of many more subactions, which, by commutation transform "a then b" into "b then a" (see (i) of Figure 1). Conversely, if actions a and b interfere (typically because of a mutual exclusion on a variable accessed by both a and b as in (ii) of Figure 1), then there is a *hole* or forbidden region, in which paths cannot enter, and then through which paths cannot be deformed. This view was first introduced by E. W. Dijkstra[3], reintroduced in a somewhat different way by V. Pratt[16] then used by J. Gunawardena[9] and after by E. Goubault[6].

Now, let us suppose we can simultaneously observe two processes acting concurrently. We are observing a *surface* now and not just a path when we were just a sequential observer, since now, our path is parameterized by two local times (see (i) of Figure 2). In some sense we are observing a schedule of the allocation of pairs of actions to pairs of processors. The "interleaving" of the three (unordered) pairs of actions (taken from the set $\{a, b, c\}$) is the boundary of the cube shown in (ii) of Figure 2, and represents the possible allocations in time of two of the three actions a, b and c on two processors, i.e. the implementation of $a \mid b \mid c$ on two processors, whereas the interior of the cube represents the purely asynchronous execution of the three actions.

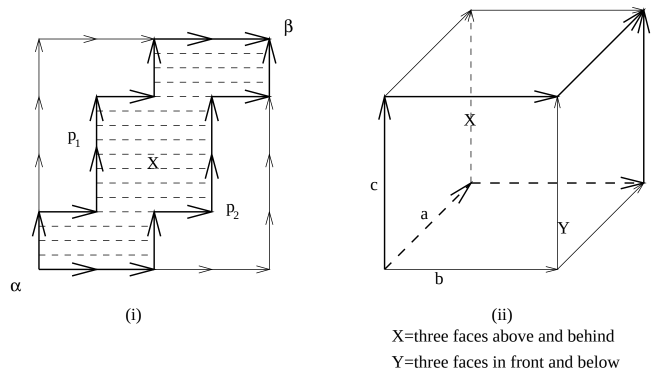

Figure 2: A concurrent execution of two processes (i) and two equivalent surfaces (ii)

There again, the two 2-schedules are homotopic if one can be deformed into the other through cubes. This can be generalized to any sequence of hypercubes. Semi-regular HDA are just amalgamated sums of points, segments, cubes and hypercubes glued along their boundaries. Basically they are just collections of n-transitions (n-cubes, i.e. asynchronous executions of n actions, abstraction of the unit hypercube $[0,1]^n$) together with two series of boundary operators. One is the series "start boundary operators" d_i^0 $(0 \leq i \leq n-1)$ which to any n transition gives one of the n $(n-1)$-transitions which can start its execution. The other one is the series of "end boundary operators" d_i^1, the obvious complement to the start boundary operators. Formally,

Definition 1 *An unlabeled semi-regular HDA is a collection of sets $M_{p,q}$ $(p,q \in \mathbb{Z})$ together with functions $d_i^0 : M_{p,q} \rightarrow M_{p-1,q}$ and $d_j^1 : M_{p,q} \rightarrow M_{p,q-1}$ for all $p,q \in \mathbb{Z}$ and $0 \leq i,j \leq p+q-1$ ($p+q$ is the dimension of the transition), such that $d_i^k \circ d_j^l = d_{j-1}^l \circ d_i^k$ ($i < j$ and $k,l = 0,1$) and $\forall n,m\ n \neq m, \quad M_n \cap M_m = \emptyset$.*

2.3 Schedules, input, output and protocol complexes

The main idea is to describe the schedulers[6] that a language or a machine can implement. Then using the detailed semantics of the shared data structures or the communication network through which the information is exchanged, we determine what amount of information might be different when we go from one schedule to another. This can then be used to decide whether a decision task can be implemented under the constraints of architecture and language. Other considerations (again of a topological nature) will be used for giving lower bounds on the complexity of the computation when shown possible.

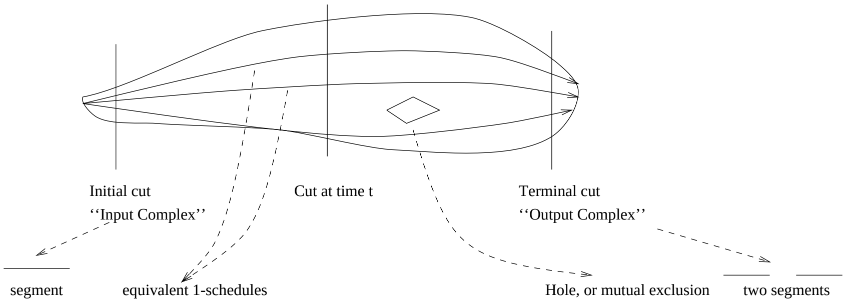

Figure 3: Geometry of schedules and cuts

The n-schedules are just the sequences of hypercubes of dimension n in an HDA. If they are homotopic then they define equivalent schedules in the sense that they will compute the same function. When we want to know whether we can implement some relation then we just know the cuts at initial time and terminal time of the HDA semantics. The equivalence of schedules induces an equivalence (a homotopy) on the different cuts. So if we can characterize geometrically the schedulers of our machine, then we will know if we can implement some relation (see Figure 3).

The problem for implementing some relations will be that the topology of the cut can never change enough. If we suppose that a relation can be computed then the new question is now, "how much time do we need to compute it?". The basic idea here is to determine at which time the topological obstruction to computing the relation is no longer here. Therefore, we need to see from what time on the topology of the cuts match the output complex.

3 Application to our asynchronous machine

We denote both the shared and local stores by ρ which is a function from $V \cup (\cup_i V_i)$ to D, the domain of values. We then extend ρ to denote all the homotopies[5] between environments. In order to retain some complexity measure on computation, we add an ordinal counter to the environment denoted by $\rho(t)$ (we suppose t is not the name of a local nor shared variable). Then, for processor i the standard denotational semantics for instructions is,

$$\begin{array}{lcl}
[\![\mathbf{nil}]\!]_d\ \rho & = & \rho[x_i \leftarrow r_i^i, t \leftarrow t+1] \\
[\![scan]\!]_d\ \rho & = & \rho[r_j^i \leftarrow x_j (j \neq i), x_i \leftarrow r_i^i, t \leftarrow t+1] \\
[\![write\ u]\!]_d\ \rho & = & \rho[x_i \leftarrow u, t \leftarrow t+1] \\
[\![r_j^i = f(r_{j1}^i, \ldots, r_{jn}^i)]\!]_d\ \rho & = & \rho[r_j^i \leftarrow f(r_{j1}^i, \ldots, r_{jn}^i), t \leftarrow t+1]
\end{array}$$

This semantics can be extended to a truly-concurrent semantics $[\![.]\!]$ using HDA by letting $[\![I]\!]\ \rho = [\rho, [\![I]\!]_d \rho]$ to be the higher-dimensional transition from ρ to $[\![I]\!]_d \rho$. Then for processes the truly-concurrent semantics is given by,

$$\begin{array}{lcl} [\![branch(r, I_1, I_2)]\!]\ \rho & = & tt^*(\rho(r))[\![I_1]\!]\ \rho + (ff^* + \perp^*)(\rho(r))[\![I_2]\!]\ \rho \\ [\![P_1; P_2]\!]\ \rho & = & [\![\hat{P_2}]\!]([\![P_1]\!]\ \rho) \end{array}$$

The equation for branch[5] is just the union of the true (tt), false (ff) and undefined ($\perp$) branches. The sequential composition is just the composition of the homotopy $[\![P_2]\!]$ with the final states of $[\![P_1]\!]\rho$, together with those of $[\![P_1]\!]\rho$. We refer to E. Goubault[5] for details of the construction of the sequentialization (via the operator $\hat{}$). Finally the semantics of the whole system is

$$[\![Prog]\!]\ \rho = \sum_{\sigma \in \mathcal{S}_n} [\![P_{\sigma(1)}]\!] \circ [\![P_{\sigma(2)}]\!] \circ \cdots \circ [\![P_{\sigma(n)}]\!]\rho$$

where $Prog = P_1 \mid \cdots \mid P_n$ and $\mathcal{S}_n$ is the group of permutations on $\{1, \cdots, n\}$.

3.1 Herlihy/Shavit/Rajsbaum's approach in a semantic framework

Notice that all equivalent schedules, or homotopic n-paths compute the same function. In order to compute a relation we need to have one schedule associated with each possible outcome from a given set of arguments. This implies that there is a direct relation between the possible schedules that a machine can express and the relations it can compute. This was used by E. Goubault[6] for the automatic verification of protocols for distributed systems. Here we want to know, given a specification of the relation we need to compute, if there is a program in $\mathcal{L}$ which implements it.

In order to do so, we will use a compact representation (which defines the simplicial complex of the decision task[10]) of the relation induced by the schedules. Actually, this could be shown to be an abstract interpretation of the concrete semantics[b]. Let X be a non-cyclic semi-regular HDA where all 1-paths from the initial states (all in $X_{0,0}$) to the final states are of the same length k. This implies that $X_{k,-k}$ is the set of final states and $X_{0,0}$ is the set of initial states. Then the cuts of X at time 0 and at time k are the codomain and domain of the relation that X computes (called respectively input and output complexes of X). More formally, if l_I is the labelling morphism which retains only the name of the process firing the transition and the value of the local variable at the source of the transition and l_O only retains the name of the process and the value of the shared variable belonging to that process

[b] It was shown[6] that the schedulers in dimension k are an abstract interpretation of the concrete HDA semantics.

then the input complex of X is the sequence $\mathcal{I} = l_I(X_{p,0}, (d_i^0)_i)_p$. The output complex of X is the sequence $\mathcal{O} = l_O(X_{k,q}, (d_i^1)_i)_q$. Notice that $\mathcal{I}$ and $\mathcal{O}$ are semi-simplicial sets.

Let $p_I : X \rightarrow \mathcal{I}$ and $p_O : X \rightarrow \mathcal{O}$ be the projections onto the initial and terminal cuts respectively (see Figure 3). The relation induced by the schedules of a program whose semantics is given by a HDA, X, is the sequence $- >= (- >_n \in \mathcal{I}_n \times \mathcal{O}_n)_n$ with $s - > s'$ if and only if, $s \in \mathcal{I}_n$ and $s' \in \mathcal{O}_n$ for some n, and there is a n-schedule u such that $p_I(u) = s$ and $p_O(u) = s'$.

4 Some results and examples on asynchronous protocols

All terms P of $\mathcal{L}$ are non-cyclic and all 1-paths from an initial state to a final state in P are of the same length, so we can use the definitions of the last section to study $\mathcal{L}$.

In the following, we will consider only protocols written in $\mathcal{L}$ such that the global store is undefined at the beginning of execution, i.e. such that the first environment is $\rho(x_i) = \perp$, $(\rho(r_i^i))_i$ being the initial tuple, argument of the program. This means that we are considering $X = [\![P_1 \mid \cdots \mid P_n]\!]\, \rho$ for which for all $s \in I_0$, $s(x_i) = \perp$. We now relate executions of the n processes with the executions of only a subset of these processes (the "solo executions"[10]). If $S = \{i_1, \cdots, i_j\} \subset \{1, \cdots, n\}$ then we write X^S, $\mathcal{I}^S$, $\mathcal{O}^S$, $- >^S$ for the HDA semantics of $P_{i_1} \mid \cdots \mid P_{i_j}$, its input complex, its output complex and its relation respectively. There are obvious inclusions between the $\mathcal{I}^S$ and $\mathcal{I}^{S'}$ when $S \subseteq S'$ (by just adding some extra local variables). Equality of states will then mean the equality on the intersection of the set of variables defined in each case. We also write $- >^{\{i\}} (s)$, $s \in \mathcal{I}_0^{\{i\}}$ for <u>the</u> point s' in $\mathcal{O}_0^{\{i\}}$ such that $s - >^{\{i\}} s'$. By induction on the semantics,

Lemma 1 *For any $S \subseteq \{1, \cdots, n\}$ such that $i, j \in S$, $i \neq j$ and any $s = \{s_i, s_j\} \in \mathcal{I}_1^{\{i,j\}}$, if $\{s_1, \cdots, s_k\}$ is the set of s' such that $s - >^S s'$ then $s_1 + \cdots + s_k$ is a 1-simplex from $- >^{\{i\}} (s_i)$ to $- >^{\{j\}} (s_j)$.*

This 1-simplex is denoted by $p(s, s')$. The different segments comprising this 1-simplex are due to the different interactions between the two processes each corresponding to different 2-schedules. These differences in the interactions can only come from *branch* instructions in the processes.

Proposition 1 *For any $S \subseteq \{1, \cdots, n\}$ such that $i_1, \cdots, i_l \in S$, the i_j being all distinct and any $s \in \mathcal{I}_l^S$, if $\{s_1, \cdots, s_k\}$ is the set of s' such that $s - >^S s'$ then $s_1 + \cdots + s_k$ is a $(l-1)$-simplex[15] whose boundaries are the l $(l-2)$-simplexes defined by the subsets $\{i_1, \cdots, \hat{i_j}, \cdots, i_l\}$.*

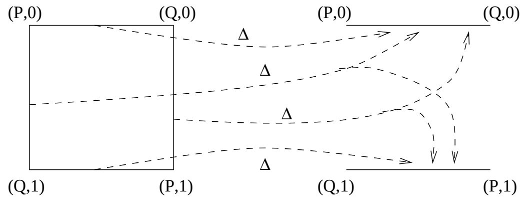

Figure 4: The specification for the binary consensus.

Theorem 1 *The image of any k-connected component of $\mathcal{I}$ ($k \leq n$) is k-connected*[15] *in $\mathcal{O}$.*

4.1 The consensus problems

The binary consensus problem will be our first example. The specification relation and input, output complexes are shown in Figure 4. An easy inpection shows that the image of the 1-simplex $((P,0),(Q,1))$ is a set of two disconnected 1-simplexes, thus violating Lemma 1. 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 synchronisation, a break of the connexity of the cuts of the dynamics. This is of course impossible in a wait-free language. Similarly, parallel or (or ordered binary consensus) cannot be implemented in a wait-free manner if the input is given locally to the processes (though there is a wait-free solution for parallel or if the input is stored in the shared memory right from the beginning).

4.2 Boolean binary relations

We first suppose that all relations R we are interested in in this section are such that $(r,\perp)R(r,\perp)$ and $(\perp,s)R(\perp,s)$. In fact, we can handle all other situations by code transformations (implementing the symmetries between processes and between values). By the theorem of the last section we know that all four segments of the input complex must be mapped onto 1-paths of the output complex, between the respective images of the vertices. We also know that the output complex must be a subcomplex of the binary 2-sphere (which is the complex left of Figure 4 for instance).

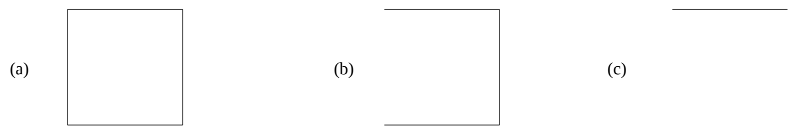

Figure 5: The three possible output complexes for wait-free binary relations

Therefore we have the three possibilities (a), (b), and (c) of Figure 5 for the output complexes. There are actually many more possibilities for the allowed relations between the input and output complexes. A typical "type (a)" program is the identity for processes P and Q. The relation in this case is therefore the identity relation on the binary 2-sphere. Notice that there are other kinds of programs of "type (a)". For instance the relation $\{(P,0),(Q,0)\}R\{(P,0),(Q,0)\}$, $\{(P,0),(Q,0)\}R\{(P,0),(Q,1)\}$, $\{(P,0),(Q,1)\}$ $R\{(P,0),(Q,1)\}$, $\{(P,0),(Q,1)\}R\{(P,0),(Q,0)\}$, $\{(P,1),(Q,1)\}R\{(P,1),(Q,1)\}$ and $\{(P,1),(Q,0)\}R\{(P,1),(Q,0)\}$ can be implemented as follows,

$$\begin{array}{rcl rcl} Prog & = & P \mid Q & \mathrm{Q} & = & scan; \\ P & = & scan; & & & branch(r_1, write(1), \mathbf{nil}) \end{array}$$

A typical "type (b)" program is pseudo-consensus. A typical "type (c)" program is composed of two constant processes in parallel.

4.3 Some complexity results

As such, we are not interested in the cost of any sequential function in particular, they are all abstracted by a single action in the semantic framework. We are in fact much more interested in the number of successive choices (i.e. *branch* statements) we have to make before reaching the solution.

Let l be the length function of 1-paths in output complexes, i.e. the number of 1-simplices which compose the 1-paths. Then we can prove that $l_{max} = max_{s,s' \in \mathcal{I}_0} l(p(s,s'))$ is the maximum number of *branch* statements. For instance, in the pseudo-consensus example, we see that the image of the 1-simplex $((P,0),(Q,1))$ is the sum of the three 1-simplexes $((P,0),(Q,0))$, $((Q,0),(P,1))$ and $((P,1),(Q,1))$. It results from an interaction of two *branch* statements.

Now, suppose that the set of possible values that each process (P or Q) can deal with is $[0,M] \cap \mathbb{Z}$ ($M \in \mathbb{Z}$). Then the input complex is the graph (V,E) with vertices $V = \{P\} \times [0,M] \cap \mathbb{Z} \cup \{Q\} \times [0,M] \cap \mathbb{Z}$ and edges $E = \{(v_1,v_2)/v_1 = (P,r), v_2 = (Q,s)\}$ with the obvious boundaries. The wait-free binary relations on that domain are actually quite constrained: all wait-free relations in $[0,M]^2$ can be computed in at most M^2 steps.

References

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

[2] 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.

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

[4] 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.

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

[6] E. Goubault. Schedulers as abstract interpretations of HDA. In *Proc. of PEPM'95*, La Jolla, June 1995. ACM Press.

[7] E. Goubault. Durations for truly-concurrent actions. In *Proceedings of ESOP'96*, number 1058. Springer-Verlag, 1996.

[8] E. Goubault. *The Geometry of Concurrency*. PhD thesis, Ecole Normale Supérieure, to be published, 1996.

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

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

[11] 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.

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

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

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

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

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

[17] 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.