# Optimal Implementation of Wait-Free Binary Relations

Eric Goubault

CNRS & LIENS, École Normale Supérieure, 45 rue d'Ulm, 75230 Paris Cedex 05, FRANCE,
email:goubault@dmi.ens.fr

**Abstract.** In this article we derive an algorithm for computing the "optimal" wait-free program on two processors that implements a given relation from the semantics of a small atomic read/write shared-memory parallel language. This algorithm is compared with the more general algorithm given in [9, 13] based on the participated set algorithm of [1]. An extension to this is given, where we add a test&set primitive to the previous language. This work is a natural follow up of [8].

## 1 Introduction and Related Work

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 two units communicating through a shared memory via atomic read/write registers (described in Section 2). This means 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 $P$, $P'$.

(a) For all integers $i$, $(i,i)\Delta(i,i)$. 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.

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

(c) For all $i$, $j$, $(i,j)\Delta(j,j)$ : $P$ 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 $\Delta$ has to be extended so that it expresses the behaviour of the system in nasty cases.

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

We should also assume (e) for all $j$, $(\perp,j)\Delta(\perp,j)$ : 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 [4], whereas the following approximate consensus, called pseudo-consensus in [9], 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. We can implement this one in a wait-free manner, as will be shown in Section 6.5.

We follow here the geometric view on distributed computation used in recent litterature in distributed protocols [2, 3, 9, 10, 11, 12, 13, 15] and in some ways in recent litterature in semantics of concurrency [6, 8, 5, 14, 17]. The idea is that wait-free relations exhibit some geometrical properties (Section 5). 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.

Not only do these relations exhibit certain properties, but conversely any relation which exhibits these properties can be constructed algorithmically at least in the case of two processors. We derive a different algorithm than the one of [9, 13] based on the participating set algorithm of [1] directly from the semantics of our language (Section 6). Its short proof stems directly from its construction. Then, after giving a few examples, we compare both algorithms (Section 7) 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 relation.

This in turn is generalized to deal with a new 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 8.

## 2 The machine and language

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 $V = \{x, x'\}$. Processor $P$ (resp. $P'$) has

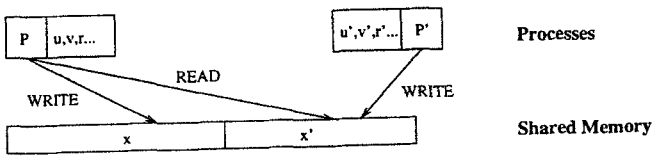



Fig. 1. Sketch of a shared memory machine with atomic read/write registers.

a local memory composed of locations $V_P = \{u, v, r \cdots\}$ (resp. $V_{P'} = \{u', v', r' \cdots\}$). 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 $x'$: SWAS or Single Write Atomic Snapshot model).

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 $P$,

$$I := update \mid scan \mid r = f(r_1, \cdots, r_n)$$

where $r, r_1, \cdots, r_n$ are local registers and $f$ is any partial recursive function.

$$\begin{array}{l} P := I \\ \quad \mid \; case\ (u_1, u_2, \ldots, u_k)\ \ of \\ \qquad\qquad (a_1^1, a_2^1, \ldots, a_k^1) : P \\ \qquad\qquad \cdots \\ \qquad\qquad (a_1^n, a_2^n, \ldots, a_k^n) : P \\ \qquad\qquad default : \quad P \\ \quad \mid \; P; P \end{array}$$

where $(u_i)_i$ are any local registers. We suppose that all tuples $(a_i^j)_i$ are different. 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 $f$ with arguments $r_1, \cdots, r_n$ and stores the result in $r$. *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.

## 3 Concrete Semantics

We denote both the shared and local stores by $\rho$ 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 $\rho$ is the value of the global and local memories at this point of the computation.

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

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

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

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

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

Otherwise,

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

We also add the obvious symmetric rules where we interchange the rôles of $P$ and $P'$. In [8], the semantics was given in terms of Higher-Dimensional Automata (HDA). This played a key rôle in giving the geometric characterization of the computable wait-free relations (to be used in Section 5). As we restricted to binary relations (i.e. to biprocessor computations) the geometric properties we need to consider are graph-theoretic properties (mainly about the number of connected components). This is why we simplified the HDA semantics to its skeleton of dimension one, i.e. the transition system generated by the rules above.

## 4 Abstraction of the Semantics

From the operational semantics of last section, we define some kind of denotational abstraction. We only retain from the concrete semantics the relation between the input value and the output value of each process. 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 2 for a picture of $S_{[1,M]\cap\mathbb{Z}}$).

- its set of vertices is $V = \{P\} \times \mathbb{Z} \cup \{P'\} \times \mathbb{Z}$,
- its set of edges is $E = \{(v_1, v_2) / v_1 = (P, r), v_2 = (P', s)\}$ with the obvious boundaries.

Following [8] we define two projections $p_I$ and $p_O$ onto $S_{\mathbb{Z}}$. $p_I$ only retains the initial value of the local variable $u$ of $P$ and $v'$ of $P'$. $p_O$ only retains the final value of $x$ for $P$ and of $x'$ for $P'$. Formally,

- if $(\{P, P'\}, \rho)$ is an initial state, $p_I(\{P, P'\}, \rho) = ((P, \rho(u)), (P', \rho(v)))$,
- if $(\{\epsilon, \epsilon\}, \rho)$ is a final state ($\epsilon$ denoting the empty string), $p_O(\{\epsilon, \epsilon\}, \rho) = ((P, \rho(x)), (P', \rho(x')))$.

The image by $p_I$ of the set of initial states for a program $\{P, P'\}$ is called the *input graph* $\mathcal{I}$. The image by $p_O$ of the set of final states is called the *output graph* $\mathcal{O}$. They are particular cases of the input complex and output complex (respectively) of [9]. They were seen as the initial and final cuts of the dynamic HDA semantics (respectively) in [8].

Now the "denotational" relation $\Delta \subseteq \mathcal{I} \times \mathcal{O}$, or *specification graph*, induced by the semantics is defined as,

$$(v_1, v_2) \Delta (v_1', v_2')$$

if and only if

- $(v_1, v_2) = p_I(\{P, P'\}, \rho)$, $(v_1', v_2') = p_O(\{\epsilon, \epsilon\}, \rho')$,
- there is a trace in the semantics of $P \mid P'$ starting at state $(\{P, P'\}, \rho)$ and ending at state $(\{\epsilon, \epsilon\}, \rho')$.

We extend the relation $\Delta$ to nodes of the graph as well. Nodes of the specification graph represent the solo executions of $P$ or $P'$. We write them as $(v_1, \perp)$ or $(P, v_1)$ for the solo execution of $P$ from state $v_1$, $(\perp, v_2)$ or $(P', v_2)$ for the solo execution of $P'$. Then $(v_1, \perp)\Delta(v_1', \perp)$ if and only if there is a solo execution of $P$ starting with private (i.e. local) state $v_1$ and ending with state $v_1'$. We have the obvious similar definition for solo executions of $P'$.

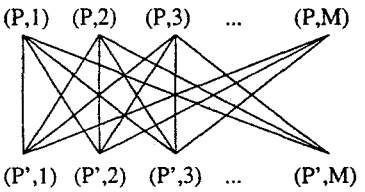



**Fig. 2.** The input graph for values in $[1, M] \cap \mathbb{Z}$.

## 5 Geometric Properties

Specification graphs represent the relation computed by programs written in our wait-free language. Conversely, given a binary relation, there is a full-abstraction problem: 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 [12]. 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 $\rho(x) = \rho(y) = \perp$ (no prior knowledge is available), then,

**Lemma 1.** *Let $\{e_1, \ldots, e_k\}$ be the image of a segment $e = ((P, u), (P', v))$ of the input graph under the relation $\Delta$, 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.*

SKETCH OF PROOF. Looking at the semantics one can prove that we can only change one value at a time (i.e. $x$ or $x'$) when exchanging information through pairs of *scan/update*, making a connected path of value changes. Formally this is proved by induction on the operational semantics. For a full proof of this, look at [7]. □

This geometric condition is satisfied for the pseudo-consensus relation as one can see by looking at the specification graph of Figure 3.

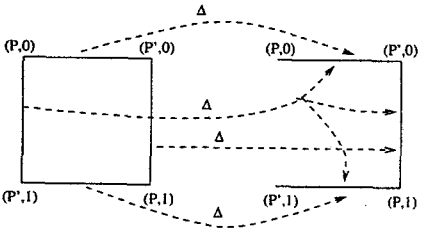



**Fig. 3.** The specification of the binary pseudo-consensus.

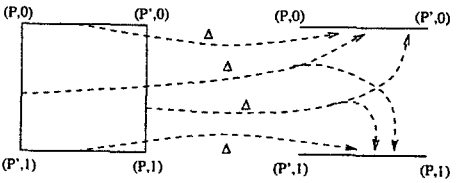


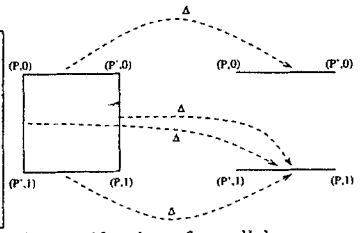

**Fig. 4.** The specification of the binary consensus.

The situation is not quite the same with binary consensus (Figure 4). An easy inspection shows that the image of the segment $((P, 0), (P', 1))$ is a set of two disconnected segments, 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 synchronization, a break of the connexity of the cuts of the dynamics [8]. This is of course impossible in a wait-free language, at least with such simple information exchanges as atomic *scan* and *update*s. Similarly, if the input is given locally to the processes as we supposed in Lemma 1, parallel or (or ordered 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:

$Prog = P \mid P'$

| $P = update; scan;$ | $P' = update; scan;$ |
|---|---|
| $case\ v\ of$ | $case\ u'\ of$ |
| $1: u = 1; update$ | $1: v' = 1; update$ |
| $default: update$ | $default: update$ |




The specification of parallel or.

## 6 Algorithmics

We will derive the algorithm from Lemma 1. First of all we will try to meet the requirements of the lemma. This will be the aim of Sections 6.1 and 6.2. Then we will find a way to describe in a recursive manner all paths $e_1, \ldots, e_k$ that appear in the lemma as image of a segment $e$. This is the aim of Section 6.3. Finally we will define the general algorithm in Section 6.4.

### 6.1 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 $\Delta$ such that $(u, \perp)\Delta(u, \perp)$ and $(\perp, v)\Delta(\perp, v)$. Suppose $(u, \perp)\Delta(f(u), \perp)$ and $(\perp, v)\Delta(\perp, g(v))$. $f$ 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 $\Delta$ if and only if $P \mid P'$ solves the specification $\Delta'$ 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 $(u, v)\Delta(u', v')$.

$$\underset{P}{P(f) = u = f(u);} \quad \underset{P'}{P'(g) = v' = f(v');}$$

SKETCH OF PROOF. 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. □

### 6.2 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 Lemma 1.

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 Lemma 1), image of $e$ by the specification relation $\Delta$. 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 5 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 6 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 7.2 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.

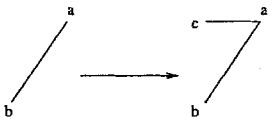



**Fig. 5.** Example of a specification graph.

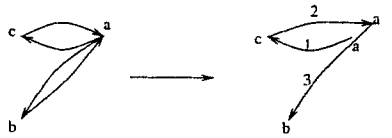



**Fig. 6.** Minimal unfolding (right) of the graph (left).

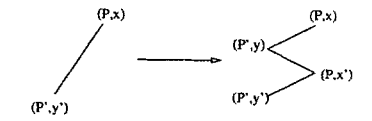
An algorithm for determining such a minimal unfolding is based on a *breadth-first* traversing strategy [16] of the graph, the traversing being complete when the criterion "having gone through all non-oriented segments and ending at $(\perp, v)$" is met. For instance, this algorithm constructs the minimal unfolding of $G_e$ which is pictured at the right of Figure 6.

### 6.3 Main code

We can now suppose that all paths image by $\Delta$ of any segment of the input graph are made of distinct segments by the unfolding of last section (one should say, oriented segments). We can also still suppose that $\Delta$ restricted to vertices is the identity relation.

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

| | |
|---|---|
| $P =$ *update*; *scan*; | $P' =$ *update*; *scan*; |
| *case* $(u, v)$ *of* | *case* $(u', v')$ *of* |
| $(x, y')$ : $u = x'$; *update*; [] | $(x, y')$ : $v' = y$; *update*; [] |
| *default* : *update* | *default* : |




Subdivision of a segment into 3 segments.

SKETCH OF PROOF. Using the semantics, we have the following three possibilities, since the only possible interactions are between the *scan* and *update* statements (the rest of the processes only act on their local memory),

(i) Suppose the *scan* operation of $P$ is completed before the *update* operation of $P'$ is started: $P$ does not know $x'$ 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* operations of $P$ and $P'$ are simulaneous. *Prog* ends up with $((P, x'), (P', y'))$.

□

*Example 1.* - The binary pseudo-consensus whose specification graph is given in Figure 3 is precisely this program with $x = 0$, $x' = 1$, $y = 0$, $y' = 1$.

- We can carry on the example specified in Figure 5, 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,

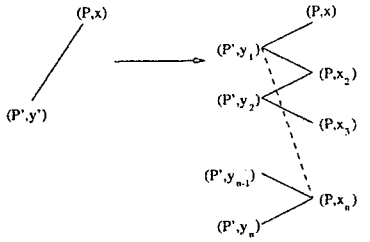
$$\begin{array}{ll} P = update; scan; & P' = update; scan; \\ \quad case\ (u, v)\ of & \quad case\ (u', v')\ of \\ \quad (x, y') :\ u = x; update & \quad (x, y') :\ v' = y; update \end{array}$$

**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 the right-hand side,

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




Subdivision of a segment into a path.

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 $y' = y_n$.

SKETCH OF PROOF. The idea is to subdivide the segment $(x_1, y_n)$ in a recursive manner (see above). 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 $n - 1$ $(x_n, y_{n-1}, \cdots, x_2, y_1)$ using $P(x_n, y_{n-1}, \cdots, x_2, y_1) \mid P'(x_n, y_{n-1}, \cdots, x_2, y_1)$. $Prog$ 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 $P'(x_n, y_{n-1}, \cdots, x_2, y_1)$ nor between $P'(x_1, y_1, x_n, y_n)$ and $P((x_n, y_{n-1}, \cdots, x_2, y_1)$. □

*Example 2.* Consider the specification graph pictured in Figure 7. The minimal unfolding is shown in two different ways in Figure 8. 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 $P' = P'(0,0,0,0)[P'(0,0,1,0)[P'(1,1,1,0)]]$.

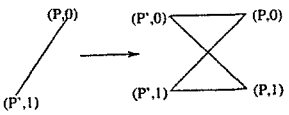



**Fig. 7.** A specification graph.

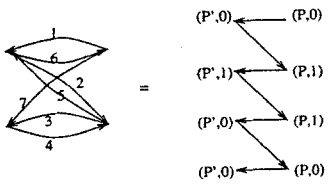



**Fig. 8.** The corresponding minimal unfolding and minimal path.

### 6.4 The 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 6.1),
- For all segments $e = ((P, u), (P', v))$ of the input graph, do,
  - determine the connected subgraph $G_e$ of the output graph, image of $e$ under the specification relation $\Delta$,
  - determine the minimal unfolding $((P, x_1) \ldots (P, x_n), (P', y_n))$ of $G_e$ (Section 6.2),
  - The program up to that point is

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

  of Section 6.3,
- 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 when applied to the processes that subdivide segments. if $(x, y') \neq (X, Y')$,

$$\begin{aligned} Mix(P(x, y, x', y')[P_1], P(X, Y, X', Y')[P_2]) = {} & update; scan; \\ & case\ (u, v)\ of \\ & (x, y') : u = x'; update; P_1 \\ & (X, Y') : v' = X'; update; P_2 \end{aligned}$$

### 6.5 Example, the binary case

As in [8] we might be interested in the case where the values of the registers are booleans, i.e. 0 or 1. There is then an easy classification theorem of all binary wait-free relations, on which we can examplify our algorithmic construction. By Lemma 1 we know that all four segments of the input graph must be mapped onto paths of the output complex, between the respective images of the vertices. We also know that the output graph must be a subgraph of the binary 2-sphere (which is the graph pictured in Figure 9).

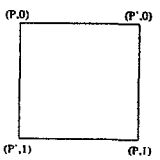



**Fig. 9.** The binary 2-sphere

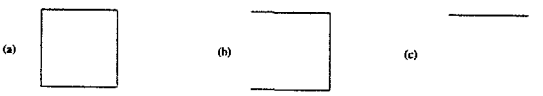



**Fig. 10.** The three possible output graphs for wait-free binary relations

Therefore we have the three possibilities (a), (b), and (c) of Figure 10 for the output graphs (up to "rotation"). 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 $P'$. The relation in this case is therefore the identity relation on the binary 2-sphere. But this is not the only relation of this type.
- Typical "type (b)" program is pseudo-consensus.
- Typical "type (c)" programs are two constant processes in parallel.

In fact all these can be seen to have a normal form of the type

$$Mix(P(0, y_0, x_0, 0), P(0, y'_0, x'_0, 1), P(1, y_1, x_1, 0), P(1, y'_1, x'_1, 1))$$

## 7 Comparison with related work

### 7.1 The participating set and Herlihy's algorithm

The participating set algorithm aims at solving the simplex agreement task of [9], that is, a generalization to any number of processors of the specification graph for pseudo-consensus.

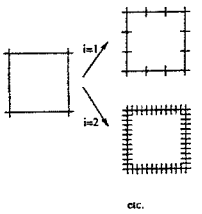



Fig. 11. Herlihy's iterated subdivision on the binary sphere.

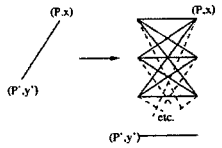



Fig. 12. The worst complexity case for a specification graph.

The intuition behind the algorithm is to subdivide all segments of the input graph, in 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 11) to create at iteration $i$ a subdivision of all segments into $3^i$ segments.

### 7.2 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 2.**

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

SKETCH OF PROOF. Looking at the algorithm of Section 6, 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 $x$, we have subdivided $e$ into a path of length $1+2x$. 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 $3max_e(s(e))$ accesses to the shared memory. In the case when all segments are mapped onto a segment except for one (like the one of Figure 12), the cost of computation is the same for all inputs and can be quite enormous. 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 12 (for all input segments). It is such that for all inputs $e$, $s(e)=t(e)$ is asymptotically $\alpha M^2$ with $\frac{1}{2}\leq\alpha\leq 1$.

SKETCH OF 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 2 to conclude. □

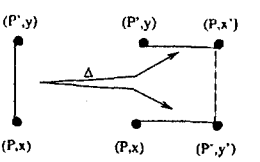
## 8 Test&Set operations

In this section we add to the language a test&set operation ($t\&s$) on a flag $f$ shared by the two processes $P$ and $P'$. 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.

**Lemma 3.** *The specification graph of the figure below can be implemented in our new language.*

SKETCH OF PROOF. The following program implements the "splitting" of one segment into two others (where ? means any value),

| | |
|---|---|
| $Prog = P \mid P'$ | |
| $P = update; scan;$ | $P' = update; scan;$ |
| $case\ (u,v,t\&s(f))\ of$ | $case\ (u',v',t\&s(f))\ of$ |
| $(x,?,0):\ u=x; update$ | $(?,y,0):\ v'=y; update$ |
| $(x,y,1):\ u=x'; update$ | $(x,y,1):\ v'=y'; update$ |



Splitting of a segment.

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. Now we can state,

**Theorem 4.** *For the SWAS model between two machines plus test&set on a shared flag, the relations $\Delta$ that can be computed are exactly the relations such that the image of any segment $(x,y)$ is a finite union of connected components, one of which contains $(P,x)$, and one of which contains $(Q,y)$.*

SKETCH OF PROOF. Basically, a given (finite) program can only split (a finite number of times) a segment and apply any subdivision on these segments. The constructive algorithm follows immediately. □

## 9 Conclusion

We have shown that wait-free binary relations could be constructed algorithmically and implemented in a small shared-memory language, giving another proof of the results of [13]. This new proof is interesting since it comes directly, through simple transformation steps and geometric intuitions, from the semantics of the language. It is also interesting since it gives an optimal implementation of these relations in terms of the number of tests and read/write operations in the main (shared) memory the processes have to execute. Numerous generalizations of this work should be considered. We have been trying to keep things as simple as possible in this article for making the main ideas clear. A straightforward generalization would be the construction of 1-resilient $n$-ary relations (i.e. relations on $n$ processors whose implementation can tolerate up to one failure of a process) since it involves the same sort of geometric phenomena on graphs. A far less straightforward generalization would be the construction of $t$-resilient $n$-ary relations with $t \geq 2$ since this involves higher-dimensional geometry.

*Acknowledgements* Many thanks to A. Venet and F. Védrine. Diagram macros from P. Taylor.

## References

1. E. Borowsky. Capturing the power of resiliency and set consensus in distributed systems. Technical report, University of California in Los Angeles, 1995.
2. E. Borowsky and E. Gafni. Generalized FLP impossibility result for $t$-resilient asynchronous computations. In *Proc. of the 25th STOC.* ACM Press, 1993.
3. 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.
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. *The Geometry of Concurrency.* PhD thesis, Ecole Normale Supérieure, 1995. to be published, 1997, also available at http://www.ens.fr/~goubault.
6. 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.
7. E. Goubault. The dynamics of wait-free distributed computations. Technical report, Research Report LIENS-96-26, December 1996.
8. 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.
9. M. Herlihy. A Tutorial on Algebraic Topology and Distributed Computation. Technical report, presented at UCLA, 1994.
10. 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.
11. M. Herlihy and S. Rajsbaum. Algebraic topology and distributed computing, a primer. Technical report, Brown University, 1995.
12. M. Herlihy and N. Shavit. The asynchronous computability theorem for $t$-resilient tasks. In *Proc. of the 25th STOC.* ACM Press, 1993.
13. M. Herlihy and N. Shavit. A simple constructive computability theorem for wait-free computation. In *Proceedings of STOC'94.* ACM Press, 1994.
14. V. Pratt. Modeling concurrency with geometry. In *Proc. of the 18th ACM Symposium on Principles of Programming Languages.* ACM Press, 1991.
15. 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.
16. B. Sedgewick. *Algorithms.* Addison-Wesley, 1988.
17. 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.