On the power of shared object types to implement one-resilient Consensus

Wai-Kau Lo*

wklo@cs.utoronto.ca

Vassos Hadzilacos†

vassos@cs.utoronto.ca

Technical Report CSRI-357

January 31, 1997

Abstract

In this paper we study the ability of shared object types to implement Consensus in asynchronous shared-memory systems where at most one process may crash. More specifically, we consider the following question: Let $n \geq 3$ and $\mathcal{S}$ be a set of object types that can be used to solve one-resilient Consensus among n processes. Can $\mathcal{S}$ always be used to solve one-resilient Consensus among $n-1$ processes? We prove that for $n = 3$ the answer is negative, even if $\mathcal{S}$ consists only of *deterministic* types. (This strengthens an earlier result by the first author proving the same fact for *nondeterministic* types.) We also prove that, in contrast, for $n > 3$ the answer to the above question is affirmative.

1 Background and overview

In this paper we consider some questions concerning fault-tolerant implementations of Consensus in asynchronous shared-memory systems. In such systems, some number of processes communicate with each other by accessing shared typed objects. Processes take steps in a completely asynchronous manner. In one step, a process may invoke an operation on a shared object. This causes the object to atomically change its state and return a response to the process invoking the operation. The new state entered by the object and the response

*Supported by a Canadian Commonwealth Scholarship.

†Supported by a grant from the Natural Sciences and Engineering Research Council of Canada, and a fellowship from the U.K. Engineering and Physical Sciences Research Council.

returned to the operation are determined by the specification of the type to which the object belongs. A process may *crash* — i.e., stop taking steps before completing its execution. A process that does not crash is called *correct*.

In this model of computation, an algorithm involving n processes is *t-resilient* if each correct process can complete any task it initiates, as long as no more than t processes crash. The algorithm is *wait-free* if it is $(n-1)$-resilient. Thus, in a wait-free algorithm, any correct process can complete its task after a finite number of its own steps regardless of the progress made by other processes. In fact, it must do so, even if all other processes crash.

In the Consensus problem, each process has a private input value and eventually may decide (irrevocably) on some value. An algorithm $\mathbf{A}$ for n processes *implements* (or *solves*) t-resilient Consensus among n processes if every execution of $\mathbf{A}$ satisfies the following properties: (i) no two processes decide different values (Agreement); (ii) the value decided by any process is the input value of some process (Validity); and (iii) if no more than t processes crash in the execution then, eventually, each correct process must decide (Termination). Consensus is a fundamental problem in fault-tolerant distributed computing. It captures the essence of many practical problems that require some form of agreement. Thus, it is important to understand when this problem is solvable.

A set of types $\mathcal{S}$ *implements* (or *solves*) t-resilient Consensus among n processes, if there is an algorithm $\mathbf{A}$ that implements t-resilient Consensus among n processes and every object used by $\mathbf{A}$ belongs to a type in $\mathcal{S}$. Throughout this paper we assume that $\mathcal{S}$ contains type $\mathsf{register}$; objects of type $\mathsf{register}$ are shared registers that support read and write operations (henceforth called simply "registers").

Let $\mathcal{S}$ be a set of types that implements t-resilient Consensus among n processes, for $n > t$. It is easy to see that $\mathcal{S}$ can be used to solve t-resilient Consensus among $n+1$ processes (and thus any number of processes greater than n). It is not clear whether $\mathcal{S}$ can also be used to solve t-resilient Consensus among $n-1$ processes (and thus any number of processes in the range $t+1$ and $n-1$). Now, reaching Consensus appears more difficult, since the algorithm can rely on fewer correct processes ($n-t-1$ instead of $n-t$). Thus, *prima facie*, it appears possible that a set of types is strong enough to solve t-resilient Consensus among n processes, but too weak to solve t-resilient Consensus among $n-1$ processes.

The question of whether this is possible was studied by Chandra *et al.* [3]. They showed that for any $t \geq 2$, any set of types strong enough to solve t-resilient Consensus among $n > t$ processes, is also strong enough to solve t-resilient Consensus among $n-1$ processes. The case of $t = 1$ was left open in that investigation. This special case is an important one since, in practice, handling a single failure is often an adequate degree of fault-tolerance [10]. In a subsequent paper, the first author considered this special case and proved that the restriction to $t \geq 2$ is necessary for the above-cited result of [3]. Specifically, in [8] he exhibited a *nondeterministic* type wor with the following property: Using only wor objects and registers, it is possible to implement one-resilient Consensus for three, but not for two, processes.

In this paper we complete the study of the special case of $t = 1$. First, we strengthen the result of [8] by exhibiting a *deterministic* type that can be used (together with registers) to implement one-resilient Consensus for three, but not two, processes. This object type has another interesting property: To our knowledge it is the only known *deterministic* type at level one of the Consensus hierarchy [5] which cannot be implemented in a one-resilient manner for three or more processes using only registers.[1] Second, and in contrast, we show that for any $n \geq 4$, a set of (deterministic or nondeterministic) types strong enough to implement one-resilient Consensus among n processes is also strong enough to implement one-resilient Consensus among $n-1$ processes.

The rest of the paper is organised as follows: In Section 2, we define formally the model of computation. In Section 3 we define the deterministic type that can be used to implement one-resilient Consensus for three, but not two, processes. In Section 4 we prove that for all $n \geq 4$, a set of types strong enough to implement one-resilient Consensus among n processes, is also strong enough to implement one-resilient Consensus among $n-1$ processes. We conclude in Section 5.

2 The model of computation

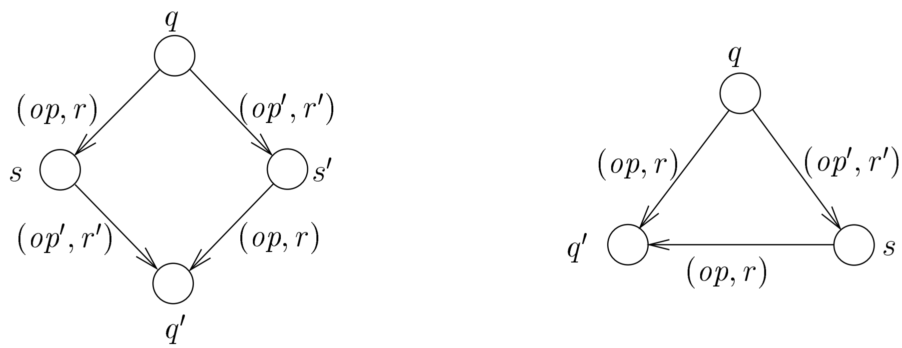
2.1 Types and objects

An *object type* T is a tuple (Q, OP, RES, δ). Q is a set of *states*, OP is a set of *operations*, RES is a set of *responses* to operations, and $\delta : Q \times OP \times RES \times Q$ is a *state transition relation*. The state transition relation of T describes the possible behaviour of an object of type T, and is sometimes called the type's *sequential specification*. Informally, $(q, op, res, q') \in \delta$ means that if the current state of the object is q and operation op is applied to it, then it is possible that res will be returned to the operation and the object will enter state q'.

We say that T is *total*, if for any $q \in Q$ and $op \in OP$, there is at least one $res \in RES$ and $q' \in Q$ such that $(q, op, res, q') \in \delta$. In other words, each operation will do something in *any* state. Every type considered in this paper is total. We say that T is *deterministic* if for any $q \in Q$ and $op \in OP$, there is at most one $res \in RES$ and $q' \in Q$ so that $(q, op, res, q') \in \delta$; otherwise, T is *nondeterministic*. If a type is deterministic (and total) δ can be viewed as a function from $Q \times OP$ to $RES \times Q$. In this case, we write $\delta(q, op) = (res, q')$ to denote that $(q, op, res, q') \in \delta$.

Let op and op' be operations of a deterministic type T, q be a state of T, and δ be the state transition relation of T. The two operations *commute* at q if applying them in either order has the same effect. More precisely, there exist responses r, r' and states s, s', q' such that $\delta(q, op) = (r, s)$, $\delta(s, op') = (r', q')$, $\delta(q, op') = (r', s')$, and $\delta(s', op) = (r, q')$; see Figure 1(a). Operation *op overwrites op'* at q, if applying op after op' at q has the same effect as applying op by itself. More precisely, there exist responses r, r' and states s, q' such that $\delta(q, op') = (r', s)$ and $\delta(q, op) = \delta(s, op) = (r, q')$; see Figure 1(b).

[1] The level of a type in the Consensus hierarchy is the maximum positive integer n such that we can implement wait-free (i.e., $(n-1)$-resilient) Consensus among n processes using only objects of that type and registers. (If there is no such maximum, the level of the type in the Consensus hierarchy is ∞.) The significance of this hierarchy lies in the following fundamental result, due to Herlihy [5]: If a type T is at level n of the Consensus hierarchy, then it is possible to use only objects of type T and registers to give a wait-free implementation of *any* object type in a system of n processes.



(a) op and op' commute at q (b) op overwrites op' at q

Figure 1: Commutative and overwriting operations.

An *object* is an instance of an object type. For the purposes of this paper, we can think of an object O of type T as an automaton whose states and state transition relation are as in T — except that states are labeled with O, to distinguish them from the states of other objects of the same type.

2.2 Processes and algorithms

A *process* is a deterministic automaton that interacts with objects. More precisely, let $\mathcal{O}$ be a set of objects, and OP, RES be the set of all operations and responses, respectively, of the types to which the objects in $\mathcal{O}$ belong. We define a *process that uses* $\mathcal{O}$ as a tuple $P = (\Sigma, \Sigma_0, \nu, \tau)$, where Σ is a set of *states*; $\Sigma_0 \subseteq \Sigma$ is a set of *initial states*; and the functions $\nu : \Sigma \to OP \times \mathcal{O}$, and $\tau : \Sigma \times RES \to \Sigma$ describe the interaction of the process with the objects. Intuitively, if P is in a state $\sigma \in \Sigma$ and $\nu(\sigma) = \langle op, O \rangle$, then in its next step P will apply operation op to object O. Based on its own current state, O will return a response res to P and will enter a new state, in accordance with the state transition relation of the type to which O belongs. Finally, P will enter state $\tau(\sigma, res)$, as a result of the response it received from O.

An *algorithm* $\mathbf{A}$ consists of a set of processes Π, a set of objects $\mathcal{O}$ so that each $P \in \Pi$ uses a subset of $\mathcal{O}$, and an initial state for each object in $\mathcal{O}$. The designated initial state of an object is one of the states of the type to which the object belongs. A *configuration* C of $\mathbf{A}$ is a tuple consisting of the state of each process in Π and each object in $\mathcal{O}$. C is an *initial* configuration of $\mathbf{A}$ if each process is in one of its initial states and each object is in the state designated as the initial state by $\mathbf{A}$.

A *step* of process P is a tuple (P, op, O, res); this indicates that P has applied operation op to object O and received response res. Let $P = (\Sigma, \Sigma_0, \nu, \tau)$ and C be a configuration, where the state of P in C is σ. If $\nu(\sigma) = \langle op, O \rangle$, we say that P has operation *op* to object O *pending* in C. If, in addition, the state of O in C is q, and *res* is a legitimate response of O to *op* in state q (i.e., for some state q' of O, (q, op, res, q') is in the state transition relation of O's type), then we say that the step $e = (P, op, O, res)$ is *applicable* to C. If $e = (P, op, O, res)$ is applicable to C, $e(C)$ denotes the configuration resulting from C after step e. More precisely, if in C the state of P is σ and the state of O is q, then $e(C)$ is the configuration in which P has state $\tau(\sigma, res)$, O is in state q' such that (q, op, res, q') is in the state transition relation of O's type, and all other processes and objects are in the same state as in C.

A *schedule* S *of algorithm* **A** is a (finite or infinite) sequence of steps of **A**'s processes. If every step in S is a step of process P, S is a *solo* schedule of P. We say that $S = e_1, e_2, \ldots, e_i, \ldots$ is applicable to a configuration C, if e_1 is applicable to C, and e_{i+1} is applicable to $e_i(e_{i-1}(\ldots(e_1(C))\ldots))$, for all i. If S is finite and has k steps, $S(C)$ denotes $e_k(e_{k-1}(\ldots(e_1(C))\ldots))$, i.e., the configuration that results after applying the steps in S one at a time, starting with configuration C. If S and S' are schedules, $S \cdot S'$ denotes their concatenation. For any configurations C and C', we say that C' is *reachable* from C, if there is a schedule S applicable to C such that $C' = S(C)$.

Let S be any infinite schedule S applicable to an initial configuration I of algorithm **A**. We say that a process P *crashes* (or is *faulty*) in S if P has only finitely many steps in S; otherwise, we say that P is *correct* in S. The idea of modeling correct processes as ones that take infinitely many steps in an infinite execution — and crashed processes as ones that take only finitely many steps — was introduced in [4]. It turns out to be a very convenient convention. We can easily accommodate processes that terminate in this framework by imagining that a process that reaches a final state takes infinitely many "do-nothing" steps thereafter.

2.3 The Consensus problem

In the Consensus problem, each process has a private initial value, drawn from the set $\{0, 1\}$. Thus, each initial configuration of a Consensus algorithm may be viewed as a function mapping each of the processes to $\{0, 1\}$. Some states of each process are associated with an irrevocable decision in $\{0, 1\}$. The decision is irrevocable in the sense that if a process enters a state associated with decision d, then all states it may subsequently enter are also associated with d. Let n, t be integers such that $n > t \geq 1$. A *t-resilient Consensus algorithm* **A** for n processes satisfies the following properties. For any initial configuration I of **A**, and any schedule S of **A** applicable to I,

Agreement: If two processes have decided in $S(I)$, then their decisions are the same.

Validity: If a process has decided in $S(I)$, then its decision must be the initial value of some process in I.

Termination: If S is infinite and at most t processes crash in S, then for any correct process P in S, there is a prefix S' of S such that P has decided in $S'(I)$.

If, in addition, $t = n - 1$, we say that $\mathbf{A}$ is a *wait-free* Consensus algorithm for n processes.

3 The (exceptional) case of three processes

In this section we introduce a deterministic type, named dor[2], that (together with registers) is strong enough to solve one-resilient Consensus for three processes, but not for two processes. The type is defined in Section 3.1. We prove that it is strong enough to solve one-resilient Consensus for three processes in Section 3.2. Finally, we prove that it is too weak to solve one-resilient Consensus for two processes (equivalently: wait-free Consensus for two processes) in Section 3.3.

3.1 Specification of dor

The specification of dor is motivated by a specific algorithm that we have in mind for implementing one-resilient Consensus among three processes using an object of this type and registers. Thus, it is best to explain dor alongside an informal description of that algorithm.

A process can apply two kinds of operations to a dor object: "enroll" an integer in the set $\{0, \ldots, 5\}$, and "query" whether 0 was enrolled before 1. In the algorithm that implements one-resilient Consensus among three processes Q_0, Q_1 and Q_2, each process enrolls two integers to a dor object O: Q_0 enrolls 0 and 3, Q_1 enrolls 1 and 4, and Q_2 enrolls 2 and 5. The three processes observe a particular protocol in enrolling their integers. Q_0 and Q_1 observe the same protocol, while Q_2 follows a slightly different one. Specifically, each of Q_0 and Q_1 enrolls its first integer in O and waits until at least one of the other two processes has also enrolled its first integer. It then enrolls its second integer and, again, waits until one of the other two processes has also done so. In contrast, Q_2 *first* waits for one of the other two processes to enroll its first integer in O and then enrolls its own first integer. Similarly, Q_2 waits for one of the other two processes to enroll its second integer and then enrolls its own second integer. Thus, Q_0 and Q_1 follow the pattern: enroll, wait, enroll, wait; while Q_2 follows the pattern: wait, enroll, wait, enroll. Note that, as a result of this protocol, the first integer to be enrolled is either 0 or 1.

After enrolling its integers in this manner, each process Q_k "queries" O to determine whether 0 was enrolled before 1. This allows the three processes to agree on which of the two integers was enrolled first — and thus to agree on the identity of one of Q_0 or Q_1, so that if they agree on Q_k, then Q_k has taken at least one step (the step required to enroll k). By requiring each of Q_0 and Q_1 to write its input value into a shared register before enrolling its first integer, we can easily turn agreement on the identity of one of these two processes into agreement on that process' input value. In this way, we can solve one-resilient Consensus among the three processes using only one dor object and some registers.

We must make special provisions to ensure that dor cannot be used to implement one-resilient Consensus for *two* processes. Informally, we shall accomplish this by ensuring that any use of a dor object that does not conform to the one described above puts the object

[2] dor stands for deterministic one-resilient.

Each node of the graph below represents a state of dor. Let δ denote the state transition relation of type dor, and q be any node of the graph shown below.

ENROLL(v) for $v \in \{0, \ldots, 5\}$: If there is an arc labeled with v from node q to node q', then $\delta(q, \text{ENROLL}(v)) = (ack, q')$. If there is not, then if $v = 5$ and q is a double-circled node, then $\delta(q, \text{ENROLL}(5)) = (ack, q)$, and otherwise, $\delta(q, \text{ENROLL}(v)) = (ack, ☹)$.

REVEAL: If q is a black node then $\delta(q, \text{REVEAL}) = (0, q)$; otherwise ($q$ is a white node), $\delta(q, \text{REVEAL}) = (1, q)$.

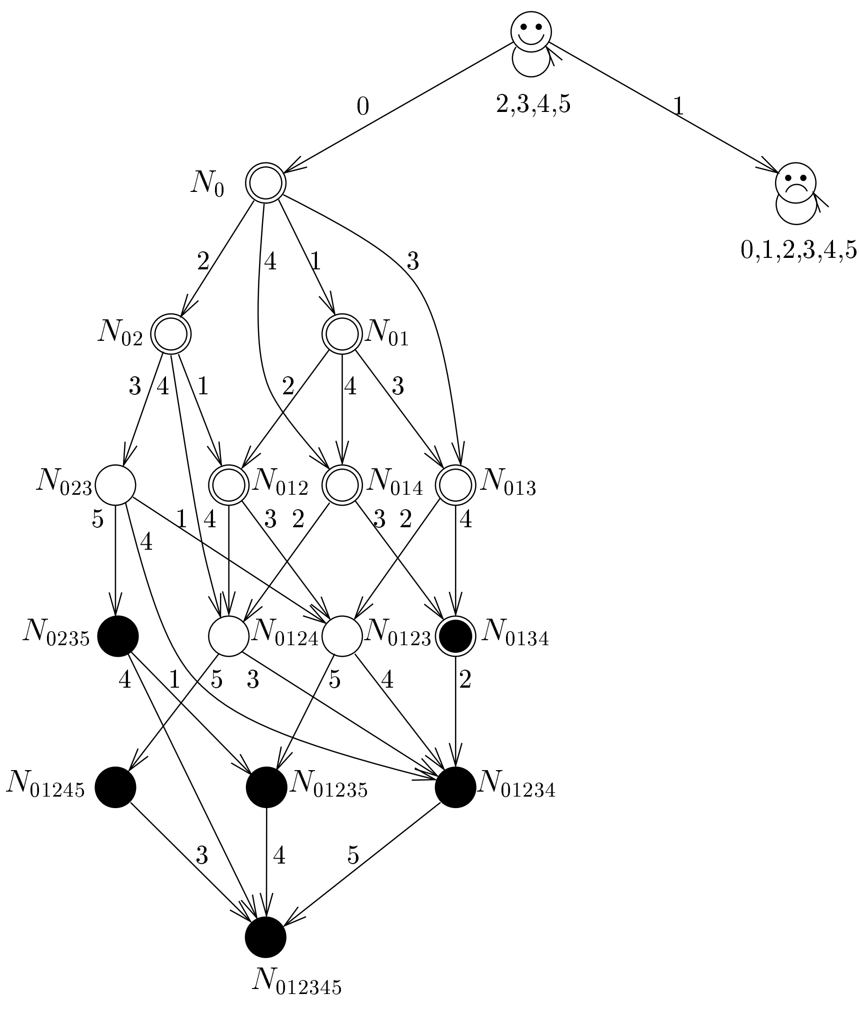


Figure 2: State transition relation of type dor.

in a state from which it is impossible to get "useful" information. In particular, it is impossible to reliably determine whether 0 or 1 was the "winner" (i.e., was enrolled first). The nondeterministic type wor described in [8] did this by defining an "upset" state that a wor object enters if not accessed as required by the algorithm described above. The object responds by nondeterministically returning 0 or 1 if queried about the winner while in that state. We cannot use this technique here because we want to define a *deterministic* type. Instead, we design the type to return deterministic, but misleading, responses if accessed "inappropriately". Doing so while ensuring that the resulting type is not strong enough to implement one-resilient Consensus for two processes turns out to be nontrivial.

We now describe the type dor in more detail. It supports the following operations: ENROLL(v), for $v \in \{0, 1, \ldots, 5\}$, and REVEAL. ENROLL($v$) is used to enroll v. It may change the state of the object but always returns the same response, *ack*. A REVEAL is used to find out if 0 was enrolled before 1. It does not change the state, and returns a value, 0 or 1. Intuitively, response 1 is supposed to indicate that 1 was enrolled before 0 (or that neither has been enrolled yet); while response 0 is supposed to indicate that 0 was enrolled before 1. Under certain conditions, however, even though 0 was enrolled before 1, the object will (misleadingly) respond 1 to such a question. This will be the case if the question is asked "prematurely", or after the object has been "mishandled". The question is premature if it is asked before at least two of $\{3, 4, 5\}$ have been enrolled. The object is mishandled if and only if, starting with the enrollment of 0, one of the following conditions takes place: (a) any number other than 5 is enrolled more than once, or (b) 5 is enrolled more than once after both 2 and one of $\{3, 4\}$ have been enrolled.

The behaviour of dor is formally specified by the state transition diagram in Figure 2. The type has seventeen states, represented as nodes in the diagram. The state labeled ☺ is called the *fresh* state; the state labeled ☹ is called the *upset* state; the remaining are called *normal* states. The colour of a node (black or white) indicates the response (0 or 1, respectively) of a REVEAL applied to the corresponding state. Recall that each ENROLL operation returns *ack*, so there is no need to explicitly represent that in the diagram.

The directed edges of the diagram represent transitions caused by ENROLL operations (recall that REVEAL operations do not affect the state). The label of an edge denotes the parameter of the corresponding ENROLL. For example, an ENROLL(0) operation applied to the fresh state causes dor to enter the normal state N_0. Note that, in each state, one or more edges (corresponding to one or more ENROLL operations) are missing. The convention regarding missing edges is as follows: Any double-circled node has an implicit transition labeled 5 to itself. Any other missing edge is an implicit transition to the upset state. Thus, for instance, node N_{01} has implicit transitions labeled 0 and 1 to the upset state, and an implicit transition labeled 5 to itself. In terms of our earlier informal description of dor, implicit transitions to the upset state correspond to operation invocations that "mishandle" the type. Thus, after such a transition has taken place, the type will (misleadingly) respond 1 to a REVEAL, even though 0 was enrolled before 1.

The fresh state is a source, in the sense that it is not reachable from any other state. The upset state is a sink, in the sense that no other state is reachable from it. Intuitively, it is the state reached if the object is initialised to the fresh state and either 1 is enrolled

Let $\mathcal{N} = \{\{0\}, \{0,1\}, \{0,2\}, \{0,1,2\}, \{0,1,3\}, \{0,1,4\}, \{0,2,3\}, \{0,1,2,3\}, \{0,1,2,4\}, \{0,1,3,4\}, \{0,2,3,5\}, \{0,1,2,3,4\}, \{0,1,2,3,5\}, \{0,1,2,4,5\}, \{0,1,2,3,4,5\}\}$.

Set of states: $\{☺, ☹\} \cup \mathcal{N}$.
Set of responses: $\{ack, 0, 1\}$.
The state transition relation of dor is defined by the two pseudocodes below:

```
ENROLL(v), v ∈ {0, ..., 5}
   case current_state
        ☹: no state change
        ☺: if v = 0 then current_state ← {0}
           else if v = 1 then current_state ← ☹
           else no state change
        N: if v ∈ N then current_state ← ☹
           else if v ∈ {1, 2} then current_state ← N ∪ {v}
           else if v = 3 then
               if 1 ∉ N and 2 ∉ N then current_state ← N ∪ {1, 3}
               else current_state ← N ∪ {3}
           else if v = 4 then
               if 1 ∉ N then current_state ← N ∪ {1, 4}
               else current_state ← N ∪ {4}
           else if v = 5 and 2 ∈ N and (3 ∈ N or 4 ∈ N) then
               current_state ← N ∪ {5}
           else no state change
   return ack
```

```
REVEAL
   case current_state
        ☹: return 1
        ☺: return 1
        N: if |N ∩ {3, 4, 5}| < 2 then return 1
           else return 0
```

Figure 3: Specification of type dor

before 0, or the object is mishandled. The normal states are the states reached if the object is initialised to the fresh state, 0 is enrolled before 1, and the object is not mishandled. In the diagram, each normal state is denoted by a node labeled $N_{v_1 \ldots v_i}$, where $\{v_1, \ldots, v_i\}$ is a nonempty subset of $\{0, 1, \ldots, 5\}$. Intuitively, the subscript of a normal state represents the set of numbers enrolled "so far", assuming the object is initialised in the fresh state. However, some enrollments are "ineffectual" (they do not affect the state), and some enrollments are "implicit" (they occur as a side-effect of other enrollments). Specifically, any attempt to enroll a number before enrolling 0 or 1 is ineffectual. (This corresponds to the self-loop at the fresh state.) Also, any attempt to enroll 5 before both 2 and one of $\{3, 4\}$ have been enrolled is ineffectual. (This corresponds to the double-circled nodes.) An implicit enrollment of 1 is forced if an attempt is made to enroll 4 before 1 has been enrolled, or to enroll 3 before both 1 and 2 have been enrolled. (This corresponds to the state transitions that "jump" a level in the diagram — e.g., the transitions labeled 3 and 4 at N_0.) Note that the subscripts of the white normal states are precisely those that contain at most one of $\{3, 4, 5\}$. In other words, they are precisely the states where at most one of $\{3, 4, 5\}$ has been (effectively) enrolled. Recall, from our earlier informal discussion of the type, that these are the states in which REVEAL returns a misleading response because the question whether 0 was enrolled before 1 is asked "prematurely".

We use the following terminology for the state transitions of type dor. Let *op* be an operation, q be a state and δ be the state transition function of dor. We say that *op* is *stationary* at q if $\delta(q, op) = (r, q)$, for some response r. In terms of Figure 3, this corresponds to the self-loop transitions in the fresh and upset states, and the implicit transitions on ENROLL(5) represented by the double-circled states. We say that *op* is *upsetting* at q if $\delta(q, op) = (r, ☹)$, for some response r. This corresponds to (implicit and explicit) transitions leading to the upset state in the diagram. We say that *op* is *ordinary* at q, if *op* is neither stationary nor upsetting at q. This corresponds to explicitly represented transitions on the left side of the diagram (i.e., all edges explicitly shown, except for the self-loops at the fresh and upset states, and the edge from the fresh to the upset state).

The reader should observe that, with very few exceptions, in each state of the type dor, any two operations either commute, or one overwrites the other. The specification of dor was crafted deliberately in this manner. As we shall see in Section 3.3, this turns out to be crucial in proving that dor cannot implement one-resilient Consensus between two processes.

An alternative formal description of dor is shown in Figure 3. The effect of each operation on the state is now described in pseudocode. The current state of the type is recorded in variable *current_state*. Each normal state is represented as a set $N \subseteq \{0, 1, \ldots, 5\}$, recording the numbers that have been effectively enrolled so far. We draw some connections between this description of dor and the preceding discussion. Premature invocations of REVEAL correspond to executions of line 3 of REVEAL. Invocations of ENROLL(v) that mishandle the type correspond to executions of line 5 of that operation. Ineffectual enrollments correspond to executions of lines 4 and 15 of ENROLL(v). Finally, implicit enrollments correspond to executions of lines 8 and 11 of ENROLL(v).

```
Shared  O: dor, initialised to state ☺
        D_0, D_1: register, each initialised to ⊥
        R_0, R_1, R_2: register, each initialised to 0
```

```
Code for process Q_k, for k ∈ {0,1}
1  D_k := initial value of Q_k
2  for ℓ := 1, 2 do
3      Apply(Q_k, ENROLL(3(ℓ − 1) + k), O)
4      R_k := ℓ
5      wait until R_k̄ ≥ ℓ or R_2 ≥ ℓ
6  i := Apply(Q_k, REVEAL, O)
7  decide D_i
```

```
Code for process Q_2
1  for ℓ := 1, 2 do
2      wait until R_0 ≥ ℓ or R_1 ≥ ℓ
3      Apply(Q_2, ENROLL(3(ℓ − 1) + 2), O)
4      R_2 := ℓ
5  i := Apply(Q_2, REVEAL, O)
6  decide D_i
```

Figure 4: A one-resilient Consensus algorithm for three processes using objects of type dor and registers.

3.2 Type dor solves one-resilient Consensus for three processes

In this subsection we prove that dor (together with registers) implements one-resilient Consensus for three processes. An algorithm that does this was described informally in the previous subsection; it is shown in pseudocode in Figure 4. The notation Apply(Q, op, O) denotes the procedure by which process Q applies operation op to object O; the procedure returns the response of the object to that operation. Register R_k, $k \in \{0,1,2\}$, is used by process Q_k to indicate how many numbers it has enrolled so far. Before enrolling their first numbers, Q_0 and Q_1 write their input values into registers D_0 and D_1, respectively.

We now turn to the correctness proof of the algorithm.

Lemma 1 *Consider any execution of the algorithm in Figure 4, in which at most one process crashes. In this execution, no correct process is stuck forever in the* **wait** *statement of its code (line 5 of Q_0 and Q_1, or line 2 of Q_2).*

PROOF. Suppose, for contradiction, that a correct process Q_k, for some $k \in \{0,1,2\}$, is stuck in the **wait** statement in the execution. There are two cases:

CASE 1. *Q_k is stuck in the first iteration of the* **for** *loop:* We first claim that $k \neq 2$. If not, then by line 2 of its code, Q_2 must always read $R_0 = 0$ and $R_1 = 0$ when it executes the **wait** statement during its first iteration of the **for** loop. This is possible only if both Q_0 and Q_1 crash, before executing line 4, during their first iteration of the **for** loop. This contradicts our hypothesis that at most one process may crash.

We next argue that $k \notin \{0,1\}$. Suppose, for contradiction, that process Q_k is stuck in the **wait** statement in line 5, for some $k \in \{0,1\}$. Then Q_k must always read $R_{\overline{k}} = 0$ and $R_2 = 0$ in executing that line.[3] Since $R_{\overline{k}}$ always equal to 0, it follows that $Q_{\overline{k}}$ crashes before

[3] Throughout this paper, $\overline{v} = 1 - v$, for any $v \in \{0,1\}$.

executing its line 4 during its first iteration of the **for** loop. Similarly, as R_2 always equal to 0, it must be that Q_2 crashes before executing its line 4 during its first iteration of the **for** loop (because we have just shown that if Q_2 is correct, it cannot be stuck forever in line 2 and hence must eventually execute line 4). Thus, both $Q_{\overline{k}}$ and Q_2 are faulty. This again contradicts the assumption that there is at most one faulty process.

CASE 2. *Q_k is stuck in its second iteration of the **for** loop:* We have just shown that no correct process can be stuck when executing its first iteration of the **for** loop. Hence, every correct process executes the second iteration of the **for** loop. Using this fact and an argument similar to the one in Case 1, we can prove that no correct process can be stuck in the **wait** statement during the second iteration of the **for** loop, as well.

Since each of these two cases leads to a contradiction, the lemma follows. □

Lemma 2 *Consider any execution of the algorithm in Figure 4. Let d be the parameter of the first* ENROLL *operation applied to the* dor *object O in that execution. Then, $d \in \{0, 1\}$ and every* REVEAL *operation applied to O during that execution returns response d.*

PROOF. By the specification of the algorithm, it can be readily verified that the first operation applied to the dor object O is either ENROLL(0) or ENROLL(1). Thus, $d \in \{0, 1\}$. We consider these two cases in turn.

CASE 1. $d = 1$: We show that whenever a REVEAL operation is applied, object O is in state ☹. Since the REVEAL operation returns 1 at state ☹, it follows that every REVEAL operation applied to O returns 1. To this end, first note that ENROLL(1) is upsetting at ☺. Since the initial state of O is ☺ and the first operation applied to O in the execution is ENROLL(1), O enters state ☹ after the first operation is applied. Since every operation is stationary at ☹, O stays in state ☹ throughout the rest of the execution. Consequently, whenever a process applies REVEAL, object O is in state ☹ and returns 1, as required.

CASE 2. $d = 0$: We need to show that every REVEAL operation applied to O gets back response 0. To do so, it suffices to show that a REVEAL operation will be applied only if O is in a black state, since the REVEAL operation returns 0 in all black states. By Figure 4, in any execution of the algorithm:

(a) for each $k \in \{0, 1, 2\}$, the sequence of operations that process Q_k applies to O is (a prefix of) ENROLL(k), ENROLL($k + 3$), REVEAL;

(b) at least two of $\{$ ENROLL(v) : $0 \leq v \leq 2 \}$ are applied before both ENROLL(3) and ENROLL(4) (because before executing its second iteration of the **for** loop, each of Q_0 and Q_1 waits for another process to apply its first ENROLL operation);

(c) either ENROLL(3) or ENROLL(4) is applied before ENROLL(5) (because before applying ENROLL(5) to O, Q_2 waits for at least one of $\{Q_0, Q_1\}$ to apply its second ENROLL operation); and

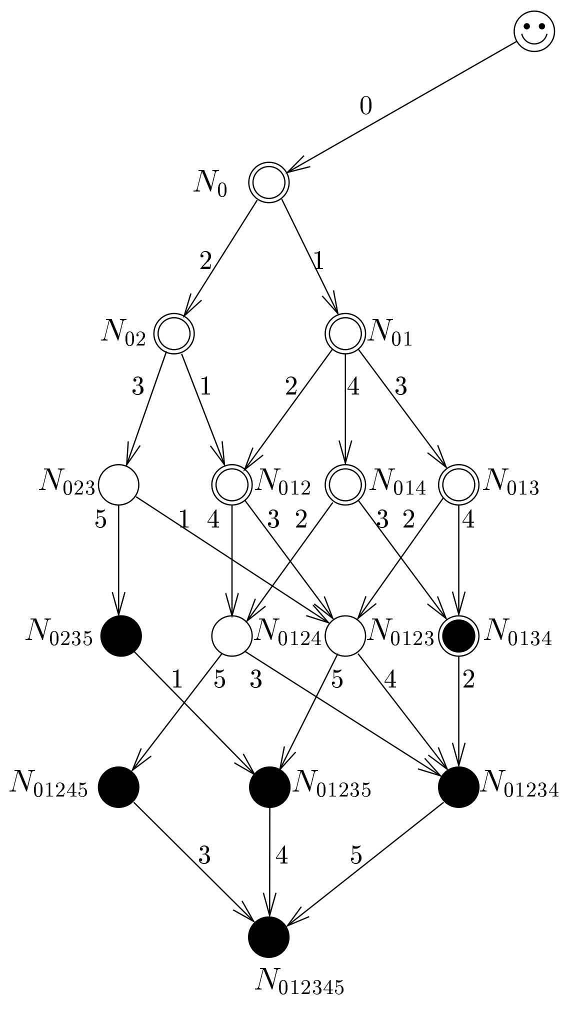


Figure 5: Possible state transitions of object O during any execution of the algorithm in Figure 4, in which the first operation applied to O is ENROLL(0).

(d) at least two of $\{\text{ENROLL}(v) : 3 \leq v \leq 5\}$ are applied before a REVEAL (because before applying REVEAL to O, every process waits for another process to apply its second ENROLL operation).

Using (a)–(c), it is easy to verify that if the first operation applied to O in the execution is ENROLL(0), then O will always be in a normal state — i.e., O never enters state ☹ — during the execution. To assist with this verification, in Figure 5 we have shown all the possible state transitions of object O in response to the ENROLL operations applied under the algorithm in Figure 4 when ENROLL(0) is the first operation applied.

By (a) and (d), when a process applies a REVEAL operation to O, at least two of the three processes have each applied exactly two ENROLL operations. By inspection of Figure 5, it is easy to see that O is in N_{0123}, N_{0124}, or a black state. (Note that N_{0123} and N_{0124} are white.) We claim that if O is in N_{0123} or N_{0124}, no process will apply REVEAL, until at least one more ENROLL operation is performed — which will put O in a black state. This is because in state N_{0123}, only Q_0 has done two ENROLL operations (with parameters 0 and 3), while each of Q_1 and Q_2 has done only one (with parameters 1 and 2, respectively). Thus, if O is in N_{0123}, by (a), the only process whose next access to O is a REVEAL is Q_0. By (d), however, Q_0 must wait until Q_1 or Q_2 has done its second ENROLL operation before it can apply REVEAL to O. A similar argument (interchanging the roles of Q_0 and Q_1) applies for state N_{0124}.

This shows that before *any* process applies REVEAL, O must have entered a black state. Furthermore, it can be immediately seen from Figure 5 that once O enters a black state, it remains in a black state throughout the execution. Thus, whenever a process applies a REVEAL, O is in a black state and thus returns 0 to the REVEAL operation, as wanted. □

Theorem 1 *The algorithm in Figure 4 is a one-resilient Consensus algorithm for three processes that uses only objects of type* dor *and registers.*

PROOF. It is clear that the algorithm in Figure 4 uses only objects of type dor and registers. For the rest of the proof, fix an arbitrary execution of the algorithm in which at most one process crashes. We need to show that this execution satisfies the three properties of Consensus — Termination, Validity and Agreement.

Termination: By Lemma 1, no correct process can be stuck in any execution of the algorithm. Therefore, every correct process decides in the execution.

Validity: By Figure 4, it is clear that, in any execution of the algorithm, only process Q_k writes into D_k, and the value Q_k writes there is its initial value, for each $k \in \{0, 1\}$. Therefore, to show that the execution satisfies Validity, it suffices to show that if a process decides the value in D_d, for some $d \in \{0, 1\}$, then Q_d has previously written into D_d. This follows from Lemma 2, since in every execution of the algorithm, a process writes into D_k before applying ENROLL(k) to O, for each $k \in \{0, 1\}$.

Agreement: Suppose that two distinct processes Q and Q' have decided in the execution (otherwise, Agreement is trivially satisfied). By Lemma 2 and the specification of the algorithm (lines 6–7 of Q_0 and Q_1, and lines 5–6 of Q_2), both Q and Q' decide the value in the same register D_d, where $d \in \{0, 1\}$ is the parameter of the first ENROLL operation applied to the dor object O during the execution. As just argued, a process reads D_d only after Q_d has written into it. Since only one value is written into D_d, Q and Q' decide the same value. Thus Agreement is satisfied. □

3.3 Type dor does not solve wait-free Consensus for two processes

In this subsection we show that, using only objects of type dor and registers, we cannot solve wait-free Consensus for two processes. (Recall that the concepts of wait-freedom and one-resilience coincide for two-process algorithms.) The argument uses a technique introduced in [8], and is outlined below.

Assume, by way of contradiction, that there exists a wait-free Consensus algorithm **A** for two processes P_0 and P_1 that uses only objects of type dor and registers. Assume, further, that **A** uses a minimal number of dor objects. (This number must be finite because the algorithm is wait-free and uses only deterministic types and there are only a finite number of different input values.) Since wait-free Consensus for two processes is not solvable using only registers [9], it follows that **A** uses at least one dor object.

To derive a contradiction, we proceed in two stages. In the first stage, we show that the assumed algorithm **A** must have a special configuration, $\mathcal{B}_0$, and a dor object, $\mathcal{O}$, with the following properties: For each $k \in \{0, 1\}$, process P_k has an ENROLL$(k + 2)$ operation to $\mathcal{O}$ pending in $\mathcal{B}_0$. Starting from $\mathcal{B}_0$, after applying ENROLL$(k + 2)$ to $\mathcal{O}$, P_k cannot apply ENROLL(v) to $\mathcal{O}$, for any $v \in \{0, 1, 2, 3\}$, and cannot decide unless it applies

- an ENROLL(4) to $\mathcal{O}$; or
- an ENROLL(5) to $\mathcal{O}$, after the other process has taken at least one step.

Furthermore, if each process is about to apply to $\mathcal{O}$ the ENROLL(4) or ENROLL(5) that it must execute before deciding, one process must be about to apply ENROLL(4) while the other must be about to apply ENROLL(5): they cannot both be about to apply ENROLL with the same parameter.

In the second stage of the proof, starting from configuration $\mathcal{B}_0$, we show how to solve Consensus for two processes without using the dor object $\mathcal{O}$. Roughly speaking, the fact that, starting from $\mathcal{B}_0$, the two processes must apply ENROLL operations to $\mathcal{O}$ with *different* parameters is exploited to satisfy Agreement, while the fact that the process that is to apply ENROLL(5) must do so in a nonsolo execution is exploited to satisfy Validity. In this way we construct a Consensus algorithm for two processes that uses one dor object less than **A**. This contradicts our hypothesis that the number of dor objects used in **A** is minimal. This contradiction means that the assumed wait-free Consensus algorithm **A** that uses only dor objects and registers does not exist.

3.3.1 Preliminaries

We first review the definition of valence for configurations of a Consensus algorithm, which was first defined in [4]. Let C be any configuration of **A**. We say that C is *bivalent* if there are configurations C_0 and C_1, reachable from C, such that some process decides 0 in C_0 and some process decides 1 in C_1. C is *v-valent*, for $v \in \{0,1\}$, if there is no configuration C' reachable from C so that some process decides $\overline{v}$ in C'. C is *univalent* if it is either 0-valent or 1-valent. We say that two univalent configurations *have the same valence* if they are both 0-valent or both 1-valent, and *have opposite valence* if one is 0-valent and the other is 1-valent. By the definitions of univalent and bivalent configuration it follows that if a configuration C of $\mathbf{A}$ is neither univalent nor bivalent, then no process decides in any configuration reachable from C. Since this contradicts the Termination property, we conclude that every configuration of a Consensus algorithm is either univalent or bivalent.

We establish some elementary properties of configurations that will be used subsequently.

Lemma 3 *Let C be any configuration of $\mathbf{A}$, and P be any process. There exists a solo schedule S of process P such that S is applicable to C and P has decided in $S(C)$.*

Proof. Immediate from the Termination requirement of Consensus and the the fact that algorithm $\mathbf{A}$ is wait-free. ☐

Lemma 4 *Let C and C' be any two univalent configurations such that the state of some process and each object is the same in C as in C'. Then, C and C' have the same valence.*

Proof. Let P be the process whose state is the same in C as in C', and S be a solo schedule of P applicable to C such that P has decided in $S(C)$. Because the state of P and each object is the same in C as in C', by induction on the length of S, we can show that S is also applicable to C'. This implies that P has decided the same value, say $v \in \{0,1\}$, in both $S(C)$ and $S(C')$. Thus, both $S(C)$ and $S(C')$ are v-valent. Since C and C' are univalent, it follows that they must also be v-valent. ☐

Lemma 5 *Let C and C' be any configurations of $\mathbf{A}$. Let S be any schedule applicable to C such that every process that takes a step in S and every object, except one* dor *object O, is in the same state in C as in C'; O is in a white state in C' and in $S'(C)$, for every prefix S' of S; and every operation applied to O in S (if any) is stationary at the state of O in C'. Then,*

- *S is applicable to C',*
- *every process that takes a step in S and every object except O is in the same state in $S(C)$ as in $S(C')$, and*
- *O is in the same state in C' as in $S(C')$.*

Proof. Let S^i be the prefix of S of length i. Recall that a REVEAL operation applied to a dor object returns 1 if the object is in a white state, and that a stationary operation applied to a dor object does not change the object's state. Using these two properties, by a straightforward induction on i, we can show that

S^i is applicable to C', every process that takes a step in S and every object except O is in the same state in $S^i(C)$ as in $S^i(C')$, and O is in the same state in C' as in $S^i(C')$.

The lemma follows immediately from this statement. □

3.3.2 Stage one of the proof

We now show the existence of a configuration and a dor object of $\mathbf{A}$ that satisfy the properties described at the beginning of Section 3.3.

Lemma 6 *There exist a bivalent configuration $\mathcal{C}$ of $\mathbf{A}$, steps e_0 and e_1 of processes P_0 and P_1, respectively, applicable to $\mathcal{C}$, and a* dor *object $\mathcal{O}$ such that*

- *$e_0(\mathcal{C})$ and $e_1(\mathcal{C})$ are univalent and have opposite valence, and*
- *e_0 and e_1 both access $\mathcal{O}$.*

PROOF. Apply the standard bivalence argument [4, 5]. □

In the rest of this section we focus exclusively on configurations that are reachable from the bivalent configuration $\mathcal{C}$. Without loss of generality, we assume that $e_0(\mathcal{C})$ is 0-valent and $e_1(\mathcal{C})$ is 1-valent. (If not, we simply interchange the roles of e_0 and e_1 throughout the argument presented below.). In what follows, let q^* denote the state of $\mathcal{O}$ in configuration $\mathcal{C}$.

Since this stage of the proof is rather long, we give a roadmap to help explain its overall structure. We first prove that q^* must be the fresh state (Lemma 10). The proof of this makes heavy use of the earlier-noted fact that dor exhibits a great deal of commutativity and overwriting between operations. We then prove that there is a solo schedule E_0 of P_0 so that P_0 has ENROLL(2) to $\mathcal{O}$ pending in $E_0(e_1(e_0(\mathcal{C})))$ (Lemma 11). In addition, there is a solo schedule E_1 of P_1 so that P_1 has ENROLL(3) to $\mathcal{O}$ pending in $E_1(E_0(e_1(e_0(\mathcal{C}))))$ (Lemma 12). Finally, we show that configuration $\mathcal{B}_0 = E_1(E_0(e_1(e_0(\mathcal{C}))))$ and dor object $\mathcal{O}$ (defined above in Lemma 6) have the properties stated at the beginning of Section 3.3 (Lemmata 15 and 16).

Lemma 7 *The two operations pending in $\mathcal{C}$ do not commute at q^*, and neither one of them overwrites the other at q^*.*

PROOF. Let op_0 and op_1 denote the (ENROLL or REVEAL) operations to $\mathcal{O}$ of P_0 and P_1, respectively, that are pending in $\mathcal{C}$.

Operations op_0 and op_1 do not commute at q^*: If not, then by definition there exist responses r_0, r_1 and states q_0, q_1, q such that, for each $k \in \{0,1\}$, $\delta(q^*, op_k) = (r_k, q_k)$ and $\delta(q_{\overline{k}}, op_k) = (r_k, q)$, where δ is the state transition relation of dor. Thus, $e_0 = (P_0, op_0, \mathcal{O}, r_0)$ and $e_1 = (P_1, op_1, \mathcal{O}, r_1)$, and they are applicable to $e_1(\mathcal{C})$ and $e_0(\mathcal{C})$, respectively. Also, $\mathcal{O}$ has the same state, namely q, in both $e_0(e_1(\mathcal{C}))$ and $e_1(e_0(\mathcal{C}))$. Thus, the state of each process and each object is in the same state in $e_0(e_1(\mathcal{C}))$ as in $e_1(e_0(\mathcal{C}))$. By Lemma 4, $e_0(e_1(\mathcal{C}))$ and $e_1(e_0(\mathcal{C}))$ have the same valence. This contradicts the fact that $e_0(e_1(\mathcal{C}))$ and $e_1(e_0(\mathcal{C}))$ have opposite valence (because $e_1(\mathcal{C})$ and $e_0(\mathcal{C})$ do).

Neither one of op_0 and op_1 overwrites the other at q^*: Suppose, for contradiction, that op_k overwrites $op_{\overline{k}}$, for some $k \in \{0,1\}$. Then, there exist responses r_0, r_1 and states s, q such that $\delta(q^*, op_{\overline{k}}) = (r_{\overline{k}}, s)$ and $\delta(q^*, op_k) = \delta(s, op_k) = (r_k, q)$, where δ is the state transition relation of dor. Thus, $e_k = (P_k, op_k, \mathcal{O}, r_k)$ and $e_{\overline{k}} = (P_{\overline{k}}, op_{\overline{k}}, \mathcal{O}, r_{\overline{k}})$, and e_k is applicable to $e_{\overline{k}}(\mathcal{C})$. After e_k is applied to $e_{\overline{k}}(\mathcal{C})$, it is easy to see that the state of P_k and each object (include $\mathcal{O}$) is the same in $e_k(e_{\overline{k}}(\mathcal{C}))$ as in $e_k(\mathcal{C})$. By Lemma 4, $e_k(e_{\overline{k}}(\mathcal{C}))$ and $e_k(\mathcal{C})$ have the same valence. This contradicts the fact that $e_k(e_{\overline{k}}(\mathcal{C}))$ is $\overline{k}$-valent (because $e_{\overline{k}}(\mathcal{C})$ is) while $e_k(\mathcal{C})$ is k-valent. □

Lemma 8 *No operation pending in $\mathcal{C}$ is stationary at q^*.*

PROOF. If, for some $k \in \{0,1\}$, the operation of P_k pending in $\mathcal{C}$ is stationary at q^*, then the operation of $P_{\overline{k}}$ pending in $\mathcal{C}$ overwrites P_k's operation at q^*. But this is not possible, by Lemma 7. □

From the specification of dor, it is easy to verify that every operation is stationary at ☹, and that the REVEAL operation is stationary at every state of dor. By Lemma 8, $q^* \neq$ ☹ and both processes have an ENROLL operation to $\mathcal{O}$ pending in $\mathcal{C}$. Therefore, e_0 and e_1 are ENROLL steps to $\mathcal{O}$, and thus are applicable to $e_1(\mathcal{C})$ and $e_0(\mathcal{C})$, respectively. For each $k \in \{0,1\}$, let v_k be the parameter of P_k's ENROLL operation pending in $\mathcal{C}$.

Lemma 9 *If q^* is normal, then*

(a) For each $k \in \{0,1\}$, ENROLL(v_k) is not upsetting at q^*.

(b) $v_0 \neq v_1$.

PROOF. (a) Suppose, for contradiction, that ENROLL(v_k) is upsetting at q^* for some $k \in \{0,1\}$. Without loss of generality, assume that $k = 0$. By Lemma 8, ENROLL(v_1) cannot be stationary in q^*, and therefore there are two remaining cases:

CASE 1. ENROLL(v_1) *is upsetting at q^**: Then, ENROLL(v_0) and ENROLL(v_1) overwrites each other at q^*. This contradicts Lemma 7.

CASE 2. ENROLL(v_1) *is ordinary at q^**: Let q be the state of $\mathcal{O}$ in $e_1(\mathcal{C})$. Since ENROLL(v_0) is upsetting at q^*, it is easy to verify from the specification of dor that ENROLL(v_0) is also upsetting at q. But then ENROLL(v_0) overwrites ENROLL(v_1) at q^*. This again contradicts Lemma 7.

Since both cases contradict Lemma 7, no ENROLL operation to $\mathcal{O}$ pending in $\mathcal{C}$ is upsetting at q^*, as wanted.

(b) Note that, for any state q of dor and $v \in \{0, \ldots, 5\}$, two ENROLL(v) operations commute at q. If, contrary to this part of the lemma, $v_0 = v_1 = v$, for some $v \in \{0, \ldots, 5\}$, the two ENROLL(v) operations pending in $\mathcal{C}$ commute at q^*, contrary to Lemma 7. Thus, $v_0 \neq v_1$. $\square$

In the rest of this section, let $\mathcal{C}_0 = e_1(e_0(\mathcal{C}))$ and $\mathcal{C}_1 = e_0(e_1(\mathcal{C}))$.

Lemma 10 $q^* = ☺$.

PROOF. By Lemma 8, $q^* \neq ☹$. Thus, it suffices to show that q^* is not normal. By Lemmata 7–9, if q^* is a normal state, the following properties hold: The two ENROLL operations pending in $\mathcal{C}$ have distinct parameters, and they are ordinary, do not commute, and neither one of them overwrites the other at q^*. By inspection of Figure 2, it can be verified that no normal state except N_0 satisfies this property. (See Figure 6 for the details of this verification.) We now eliminate the remaining possibility, i.e., that $q^* = N_0$.

By the specification of dor, ENROLL(v) is ordinary at N_0 for $v \in \{1, 2, 3, 4\}$. Thus, by Lemma 8 and Lemma 9(a), $v_0, v_1 \in \{1, 2, 3, 4\}$. First we prove that $v_0, v_1 \neq 1$. Suppose to the contrary that $v_k = 1$ for some $k \in \{0, 1\}$. Then, by Lemma 9(b), $v_{\overline{k}} \in \{2, 3, 4\}$. By inspection of the graph in Figure 2, at state N_0, ENROLL(2) and ENROLL(1) commute, ENROLL(3) overwrites ENROLL(1), and ENROLL(4) overwrites ENROLL(1). Each of these three cases contradicts Lemma 7. Hence, $v_0, v_1 \neq 1$.

Next we show that $v_0, v_1 \neq 4$. Again suppose, for contradiction, that $v_k = 4$ for some $k \in \{0, 1\}$. Then $v_{\overline{k}} \in \{2, 3\}$ (by Lemma 9(b) and the claim in the preceding paragraph). By inspection of Figure 2, ENROLL(4) commutes with both ENROLL(2) and ENROLL(3) at N_0. This contradicts Lemma 7.

We have shown that $v_0, v_1 \notin \{1, 4\}$. Thus, $v_0, v_1 \in \{2, 3\}$ and, by Lemma 9(b), they are distinct. Without loss of generality assume that $v_0 = 2$ and $v_1 = 3$. (The other case, where $v_0 = 3$ and $v_1 = 2$, is similar.) Thus, $e_0 = (P_0, \text{ENROLL}(2), \mathcal{O}, ack)$ and $e_1 = (P_1, \text{ENROLL}(3), \mathcal{O}, ack)$. Note that $\mathcal{C}_0$ and $\mathcal{C}_1$ are identical, except the state of $\mathcal{O}$ in the former is N_{023} while in the latter it is N_{0123}.

Claim 1 *There is a schedule $\hat{S}$ such that (a) $\hat{S}$ is applicable to both $\mathcal{C}_0$ and $\mathcal{C}_1$ (b) the state of each process and each object except $\mathcal{O}$ is the same in $\hat{S}(\mathcal{C}_0)$ as in $\hat{S}(\mathcal{C}_1)$, (c) each process has an* ENROLL *operation to $\mathcal{O}$ pending in both $\hat{S}(\mathcal{C}_0)$ and $\hat{S}(\mathcal{C}_1)$, and (d) the state of $\mathcal{O}$ in $\hat{S}(\mathcal{C}_0)$ is N_{023} while in $\hat{S}(\mathcal{C}_1)$ it is N_{0123}.*

PROOF. By Lemma 3, P_0 has a solo schedule S applicable to $\mathcal{C}_0$ such that P_0 has decided in $S(\mathcal{C}_0)$. Since $\mathcal{C}_0$ is 0-valent, P_0 decides 0 in $S(\mathcal{C}_0)$. We claim that S has an ENROLL step to $\mathcal{O}$. If not, since N_{023} and N_{0123} (the states of $\mathcal{O}$ in $\mathcal{C}_0$ and $\mathcal{C}_1$, respectively) are both white and since REVEAL (the only other kind of operation that may be applied to $\mathcal{O}$) is stationary at these states, then by Lemma 5, S is applicable to $\mathcal{C}_1$ and P_0 has the same state in $S(\mathcal{C}_0)$ as in $S(\mathcal{C}_1)$. This implies P_0 also decides 0 in $S(\mathcal{C}_1)$, which contradicts the fact that $\mathcal{C}_1$ is 1-valent. Thus, S contains an ENROLL step to $\mathcal{O}$, as claimed.

Let S_0 be the longest prefix of S that does not contain an ENROLL step to $\mathcal{O}$. Again by Lemma 5, S_0 is applicable to $\mathcal{C}_1$, the state of each process and each object except $\mathcal{O}$ is the

For each $v \in \{1,2,3,4\}$, ENROLL(v) is ordinary at N_V if $v \notin V$. Also, ENROLL(5) is ordinary at N_V if $5 \notin V$ and 2 and at least one of $\{3,4\}$ are in V.

1. N_{01}: Only ENROLL(2), ENROLL(3) and ENROLL(4) are ordinary at N_{01}, and any two of these three commute.
2. N_{02}: Only ENROLL(1), ENROLL(3) and ENROLL(4) are ordinary at N_{02}; and ENROLL(4) overwrites ENROLL(1), while ENROLL(3) commutes with ENROLL(1) and ENROLL(4).
3. N_{012}: Only ENROLL(3) and ENROLL(4) are ordinary at N_{012}, and they commute.
4. N_{013}: Only ENROLL(2) and ENROLL(4) are ordinary at N_{013}, and they commute.
5. N_{014}: Only ENROLL(2) and ENROLL(3) are ordinary at N_{014}, and they commute.
6. N_{023}: Only ENROLL(1), ENROLL(4) and ENROLL(5) are ordinary at N_{023}; and ENROLL(4) overwrites ENROLL(1), while ENROLL(5) commutes with ENROLL(1) and ENROLL(4).
7. N_{0123}: Only ENROLL(4) and ENROLL(5) are ordinary at N_{0123}, and they commute.
8. N_{0124}: Only ENROLL(3) and ENROLL(5) are ordinary at N_{0124}, and they commute.
9. N_{0134}: Only ENROLL(2) is ordinary at N_{0134}.
10. N_{0235}: Only ENROLL(1) and ENROLL(4) are ordinary at N_{0235}, and ENROLL(4) overwrites ENROLL(1).
11. N_{01234}: Only ENROLL(5) is ordinary at N_{01234}.
12. N_{01235}: Only ENROLL(4) is ordinary at N_{01235}.
13. N_{01245}: Only ENROLL(3) is ordinary at N_{01245}.
14. N_{012345}: There is no ordinary ENROLL operation at N_{012345}.

Figure 6: Normal states that cannot be q^*.

same in $S_0(\mathcal{C}_0)$ as in $S_0(\mathcal{C}_1)$, and the state of $\mathcal{O}$ in $S_0(\mathcal{C}_0)$ is N_{023} while in $S_0(\mathcal{C}_1)$ it is N_{0123}. Clearly, by the definition of S_0, the operation of P_0 pending in both $S_0(\mathcal{C}_0)$ and $S_0(\mathcal{C}_1)$ is an ENROLL to $\mathcal{O}$.

Applying a similar argument (replacing P_0 with P_1) to configurations $S_0(\mathcal{C}_0)$ and $S_0(\mathcal{C}_1)$, we can show that there exists a solo schedule S_1 of P_1 such that S_1 is applicable to both $S_0(\mathcal{C}_0)$ and $S_0(\mathcal{C}_1)$, the state of each process and each object except $\mathcal{O}$ is the same in $S_1(S_0(\mathcal{C}_0))$ as in $S_1(S_0(\mathcal{C}_1))$, P_1 has an ENROLL to $\mathcal{O}$ pending in both $S_1(S_0(\mathcal{C}_0))$ and $S_1(S_0(\mathcal{C}_1))$, and the state of $\mathcal{O}$ in $S_1(S_0(\mathcal{C}_0))$ is N_{023} while in $S_1(S_0(\mathcal{C}_1))$ it is N_{0123}. Recall that P_0 has an ENROLL to $\mathcal{O}$ pending in $S_0(\mathcal{C}_0)$ and $S_0(\mathcal{C}_1)$. Since S_1 is a solo schedule of P_1, the operation of P_0 pending in both $S_1(S_0(\mathcal{C}_0))$ and $S_1(S_0(\mathcal{C}_1))$ is still an ENROLL to $\mathcal{O}$. Thus, $\hat{S} = S_0 \cdot S_1$ has the properties stated in the claim. □ Claim 1

Let $A_0 = \hat{S}(\mathcal{C}_0)$ and $A_1 = \hat{S}(\mathcal{C}_1)$, and let w_0 and w_1 be the parameters of the ENROLL operations of P_0 and P_1, respectively, pending in both A_0 and A_1. For each $k \in \{0, 1\}$, let $c_k = (P_k, \text{ENROLL}(w_k), \mathcal{O}, \mathit{ack})$; i.e., c_k is the step of P_k that is applicable to both A_0 and A_1. There are four cases:

CASE 1. *$w_k \in \{0, 2, 3\}$ for some $k \in \{0, 1\}$*: Then ENROLL(w_k) is upsetting in both N_{023} and N_{0123} (see Figure 2). Since $\mathcal{O}$ has states N_{023} and N_{0123} in configurations A_0 and A_1, respectively, it follows that $\mathcal{O}$ is in state ☹ in both $c_k(A_0)$ and $c_k(A_1)$. Then, the state of P_k and each object (including $\mathcal{O}$) in $c_k(A_0)$ is the same as in $c_k(A_1)$. By Lemma 4, $c_k(A_0)$ and $c_k(A_1)$ have the same valence. This contradicts the fact that $c_k(A_0)$ is 0-valent and $c_k(A_1)$ is 1-valent.

CASE 2. *$w_k = 4$ for some $k \in \{0, 1\}$*: In this case, the state of $\mathcal{O}$ in both $c_k(A_0)$ and $c_k(A_1)$ is N_{01234} (see Figure 2). Thus, the state of P_k and each object is the same in $c_k(A_0)$ as in $c_k(A_1)$. By Lemma 4, $c_k(A_0)$ and $c_k(A_1)$ have the same valence. which is a contradiction as in Case 1.

CASE 3. *$w_k = 1$, for some $k \in \{0, 1\}$*: In this case, $\mathcal{O}$ is in the same state, namely N_{0123}, in both $c_k(A_0)$ and A_1 (see Figure 2). The state of P_k and each object is the same in $c_k(A_0)$ as in A_1. By Lemma 4, $c_k(A_0)$ and A_1 have the same valence. This contradicts the fact that $c_k(A_0)$ is 0-valent while A_1 is 1-valent.

CASE 4. *$w_0 = w_1 = 5$*: Then, $\mathcal{O}$ is in the same state, namely ☹, in both $c_1(c_0(A_0))$ and $c_1(c_0(A_1))$ (see Figure 2). It follows that every process and every object is in the same state in $c_1(c_0(A_0))$ as in $c_1(c_0(A_1))$. This contradicts Lemma 4, since the nodes $c_1(c_0(A_0))$ and $c_1(c_0(A_1))$ have opposite valence.

Each of these cases leads to a contradiction. Thus $q^* \neq N_0$, and the lemma follows. □

By the specification of dor (see Figure 2), every operation — except ENROLL(0) and ENROLL(1) — is stationary at ☺. Thus, by Lemma 8, the two operations to $\mathcal{O}$ pending in $\mathcal{C}$ must be either ENROLL(0) or ENROLL(1). Furthermore, one of these must be ENROLL(0), while the other is ENROLL(1). (If not, since two ENROLL(0)'s or two ENROLL(1)'s commute at ☺, then the two operations pending in $\mathcal{C}$ commute at ☺, which contradicts Lemma 7.) Without loss of generality, we assume that P_0 and P_1 have $\textsc{enroll}(0)$ and $\textsc{enroll}(1)$ to $\mathcal{O}$, respectively, pending in $\mathcal{C}$. (If not, we simply interchange the roles of P_0 and P_1, as well as of e_0 and e_1, throughout the argument presented below.) Thus, $e_0 = (P_0, \textsc{enroll}(0), \mathcal{O}, \mathit{ack})$ and $e_1 = (P_1, \textsc{enroll}(1), \mathcal{O}, \mathit{ack})$, and therefore $\mathcal{C}_0$ and $\mathcal{C}_1$ differ only in that the state of $\mathcal{O}$ in $\mathcal{C}_0$ is N_{01} while in $\mathcal{C}_1$ it is ☹.

Lemma 11 *There exists a solo schedule E_0 of process P_0 such that*

- *E_0 is applicable to both $\mathcal{C}_0$ and $\mathcal{C}_1$;*
- *$E_0(\mathcal{C}_0)$ and $E_0(\mathcal{C}_1)$ are identical, except the state of $\mathcal{O}$ in $E_0(\mathcal{C}_0)$ is N_{01}, while in $E_0(\mathcal{C}_1)$ it is ☹; and*
- *P_0 has an $\textsc{enroll}(2)$ to $\mathcal{O}$ pending in both $E_0(\mathcal{C}_0)$ and $E_0(\mathcal{C}_1)$.*

Proof. Consider the configuration $e_0(\mathcal{C})$. By Lemma 3, P_0 has a solo schedule S applicable to $e_0(\mathcal{C})$ such that P_0 has decided in $S(e_0(\mathcal{C}))$. Since $e_0(\mathcal{C})$ is 0-valent, P_0 decides 0 in $S(e_0(\mathcal{C}))$. (See Figure 7.) We claim that S contains an $\textsc{enroll}(v)$ step to $\mathcal{O}$ for some $v \in \{0, \ldots, 4\}$. To prove this claim we suppose, for contradiction, that S does not contain such an ENROLL step. First, note that process P_0 and each object except $\mathcal{O}$ is in the same state in $e_0(\mathcal{C})$ as in $\mathcal{C}_1$ (recall that $\mathcal{C}_1 = e_0(e_1(\mathcal{C}))$). Also, the state of $\mathcal{O}$ in $e_0(\mathcal{C})$ is N_0 while in $\mathcal{C}_1$ it is ☹; thus, $\mathcal{O}$ is in a white state in both $e_0(\mathcal{C})$ and $\mathcal{C}_1$. Since $\textsc{enroll}(5)$ and REVEAL (the only operations that may be applied to $\mathcal{O}$ in S) are stationary at both N_0 and ☹, it follows from Lemma 5 (with $C = e_0(\mathcal{C})$ and $C' = \mathcal{C}_1$) that S is applicable to $\mathcal{C}_1$ and P_0 has the same state in $S(e_0(\mathcal{C}))$ as in $S(\mathcal{C}_1)$. This implies that P_0 also decides 0 in $S(\mathcal{C}_1)$, which contradicts the fact that $\mathcal{C}_1$ is 1-valent. Hence, S contains an $\textsc{enroll}(v)$ step to $\mathcal{O}$, for some $v \in \{0, \ldots, 4\}$, as claimed.

Let $E_0 \cdot d$ be the shortest prefix of S that contains an ENROLL step to $\mathcal{O}$ other than $\textsc{enroll}(5)$. Thus, $d = (P_0, \textsc{enroll}(v), \mathcal{O}, \mathit{ack})$ for some $v \in \{0, \ldots, 4\}$, and E_0 contains no ENROLL step to $\mathcal{O}$ except $\textsc{enroll}(5)$. Since N_0, N_{01} and ☹ (the states of $\mathcal{O}$ in $e_0(\mathcal{C})$, $\mathcal{C}_0$ and $\mathcal{C}_1$, respectively) are white and both $\textsc{enroll}(5)$ and REVEAL (the only operations that may be applied to $\mathcal{O}$ in E_0) are stationary at these states, it follows from Lemma 5 (with $C = e_0(\mathcal{C})$, $C' \in \{\mathcal{C}_0, \mathcal{C}_1\}$, and $S = E_0$) that

- E_0 is applicable to both $\mathcal{C}_0$ and $\mathcal{C}_1$,
- the state of P_0 and each object except $\mathcal{O}$ is the same in $E_0(\mathcal{C}_0)$ as in $E_0(\mathcal{C}_1)$, and
- the state of $\mathcal{O}$ in $E_0(\mathcal{C}_0)$ is N_{01}, while in $E_0(\mathcal{C}_1)$ it is ☹.

Clearly, P_1 has the same state in $E_0(\mathcal{C}_0)$ as in $E_0(\mathcal{C}_1)$ (since E_0 is a solo schedule of P_0). To complete the proof of this lemma, it remains to show that $v = 2$. For this, it suffices to preclude the possibility that $v \in \{0, 1, 3, 4\}$. Let $A_0 = d(E_0(e_0(\mathcal{C})))$ and $A_1 = d(E_0(\mathcal{C}_1))$.

For each $k \in \{0,1\}$, $e_k = (P_k, \textsc{enroll}(k), \mathcal{O}, ack)$.

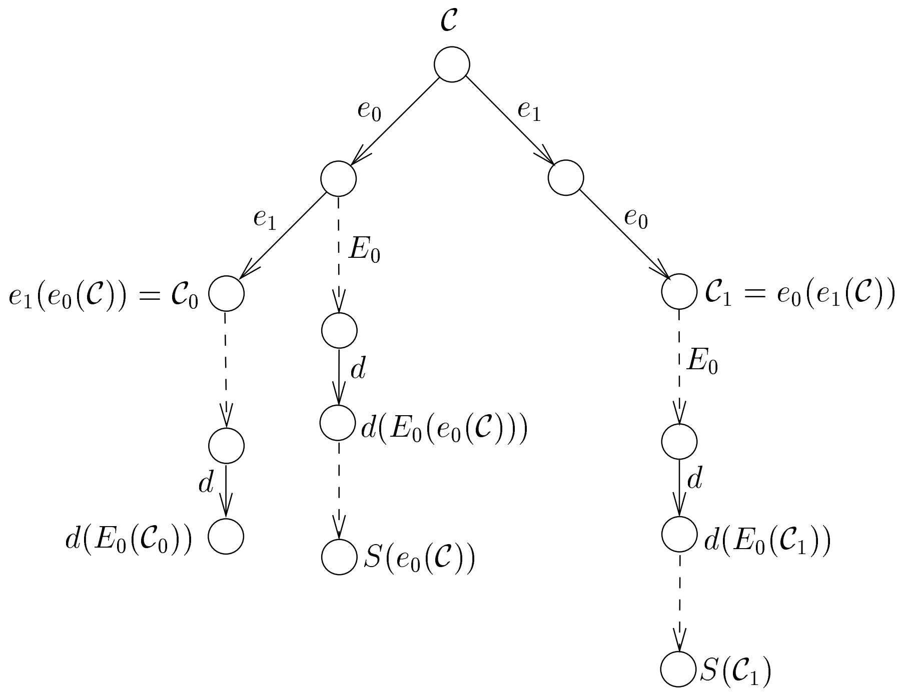


Figure 7: A solo schedule S of P_0 that is applicable to $\mathcal{C}$.

Case 1. $v = 0$: Since $\mathcal{O}$ is in state N_0 in $E_0(e_0(\mathcal{C}))$ and $\textsc{enroll}(0)$ is upsetting at N_0 (see Figure 2), $\mathcal{O}$ is in state ☹ in A_0. Since $\mathcal{O}$ is in state ☹ in $E_0(\mathcal{C}_1)$, it remains in that state in A_1. Thus, the state of P_0 and each object is the same in A_0 as in A_1. By Lemma 4, A_0 and A_1 have the same valence. This is a contradiction, since A_0 and A_1 have opposite valence.

Case 2. $v \in \{1,3,4\}$: Since $\mathcal{O}$ has state N_0 in $E_0(e_0(\mathcal{C}))$, the state of $\mathcal{O}$ in A_0 is N_{01} if $v = 1$, N_{013} if $v = 3$, and N_{014} if $v = 4$ (see Figure 2). Recall that P_1 has an $\textsc{enroll}(1)$ to $\mathcal{O}$ pending in $e_0(\mathcal{C})$. Consequently, as $E_0 \cdot d$ is a solo schedule of P_0, the operation of P_1 pending in A_0 is still $\textsc{enroll}(1)$ to $\mathcal{O}$. Thus, $e_1 = (P_1, \textsc{enroll}(1), \mathcal{O}, ack)$ is the step of P_1 that is applicable to A_0. By the specification of dor, $\textsc{enroll}(1)$ is upsetting at each of $\{N_{01}, N_{013}, N_{014}\}$. It follows that $\mathcal{O}$ is in state ☹ in $e_1(A_0)$. Then, each process and object (including $\mathcal{O}$) has the same state in $e_1(A_0)$ as in A_1. By Lemma 4, $e_1(A_0)$ and and A_1 have the same valence. This contradicts the fact that $e_1(A_0)$ is 0-valent and A_1 is 1-valent. □

Lemma 12 *Let E_0 be as in Lemma 11. There exists a solo schedule E_1 of process P_1 such that*

- *E_1 is applicable to $E_0(\mathcal{C}_0)$ and $E_0(\mathcal{C}_1)$;*
- *$\mathcal{B}_0 = E_1(E_0(\mathcal{C}_0))$ and $\mathcal{B}_1 = E_1(E_0(\mathcal{C}_1))$ are identical, except the state of $\mathcal{O}$ in $\mathcal{B}_0$ is N_{01} while in $\mathcal{B}_1$ it is $\frownie$; and*
- *for some $v \in \{3, 4\}$, P_1 has an* ENROLL*(v) to $\mathcal{O}$ pending in both $\mathcal{B}_0$ and $\mathcal{B}_1$.*

PROOF. Consider the configuration $E_0(\mathcal{C}_0)$. By Lemma 3, process P_1 has a solo schedule S_1 applicable to $E_0(\mathcal{C}_0)$ such that P_1 has decided in $S(E_0(\mathcal{C}_0))$. By Lemma 5, using an argument similar to that in Lemma 11, we can show that S has a prefix $E_1 \cdot d$ such that $d = (P_1, \text{ENROLL}(v), \mathcal{O}, ack)$ for some $v \in \{0, \ldots, 4\}$, E_1 is applicable to both $E_0(\mathcal{C}_0)$ and $E_0(\mathcal{C}_1)$, and $\mathcal{B}_0 = E_1(E_0(\mathcal{C}_0))$ and $\mathcal{B}_1 = E_1(E_0(\mathcal{C}_1))$ are identical, except that the state of $\mathcal{O}$ in $\mathcal{B}_0$ is N_{01} while in $\mathcal{B}_1$ it is $\frownie$. To complete the proof it remains to show that $v \in \{3, 4\}$. For this, it suffices to preclude the possibility that $v \in \{0, 1, 2\}$.

CASE 1. $v \in \{0, 1\}$: Since $\mathcal{O}$ is in state N_{01} in $\mathcal{B}_0$ and both ENROLL(0) and ENROLL(1) are upsetting at N_{01} (see Figure 2), object $\mathcal{O}$ is in state $\frownie$ in $d(\mathcal{B}_0)$. Clearly, as $\mathcal{O}$ is in state $\frownie$ in $\mathcal{B}_1$, it remains in state $\frownie$ in $d(\mathcal{B}_1)$. Therefore, each process and object is in the same state in $d(\mathcal{B}_0)$ as in $d(\mathcal{B}_1)$. By Lemma 4, $d(\mathcal{B}_0)$ and $d(\mathcal{B}_1)$ have the same valence. This contradicts the fact that $d(\mathcal{B}_0)$ is 0-valent and $d(\mathcal{B}_1)$ is 1-valent.

CASE 2. $v = 2$: In this case, $\mathcal{O}$ is in state N_{012} in $d(\mathcal{B}_0)$ (see Figure 2). By Lemma 11, P_0 has ENROLL(2) to $\mathcal{O}$ pending in both $E_0(\mathcal{C}_0)$ and $E_0(\mathcal{C}_1)$. Since $E_1 \cdot d$ is a solo schedule of P_1, the operation of P_0 pending in both $d(\mathcal{B}_0)$ and $d(\mathcal{B}_1)$ is still ENROLL(2) to $\mathcal{O}$. Let $d' = (P_0, \text{ENROLL}(2), \mathcal{O}, ack)$ denote the step of P_0 that is applicable to both $d(\mathcal{B}_0)$ and $d(\mathcal{B}_1)$.

Since ENROLL(2) is upsetting at N_{012} (the state of $\mathcal{O}$ in $d(\mathcal{B}_0)$), it follows that $\mathcal{O}$ is in state $\frownie$ in $d'(d(\mathcal{B}_0))$. Because $\mathcal{O}$ is in state $\frownie$ in $e_1(\mathcal{C})$, it is also in that state in $d'(d(\mathcal{B}_1))$. Thus, each process and object is in the same state in $d'(d(\mathcal{B}_0)$ as in $d'(d(\mathcal{B}_1))$. By Lemma 4, $d'(d(\mathcal{B}_0))$ and $d'(d(\mathcal{B}_1))$ have the same valence. This contradicts the fact that $d'(d(\mathcal{B}_0))$ is 0-valent and $d'(d(\mathcal{B}_1))$ is 1-valent. $\square$

By Lemma 12, we can assume, without loss of generality, that P_1 has ENROLL(3) to $\mathcal{O}$ pending in both $\mathcal{B}_0$ and $\mathcal{B}_1$. (If the operation of P_1 to $\mathcal{O}$ pending in $\mathcal{B}_0$ and $\mathcal{B}_1$ is an ENROLL(4), instead of ENROLL(3), we interchange the roles of ENROLL(3) and ENROLL(4), and replace states N_{013} and N_{0123} with states N_{014} and N_{0124}, respectively, throughout the argument presented below.)

To summarise, $\mathcal{B}_0$ and $\mathcal{B}_1$ satisfy the following properties: They are 0-valent and 1-valent configurations, respectively, reachable from $\mathcal{C}$ and differ only in that the state of $\mathcal{O}$ in $\mathcal{B}_0$ is N_{01} while in $\mathcal{B}_1$ it is $\frownie$. Furthermore, for each $k \in \{0, 1\}$, process P_k has an ENROLL$(k+2)$ to $\mathcal{O}$ pending in both $\mathcal{B}_0$ and $\mathcal{B}_1$. (See Figure 8.)

We now prove some properties of schedules that are applicable to $\mathcal{B}_0$.

Lemma 13 *Let S be any schedule that is applicable to $\mathcal{B}_0$. For each $k \in \{0, 1\}$, the first step taken by P_k in S is an* ENROLL*$(k+2)$ to object $\mathcal{O}$.*

For each $k \in \{0,1\}$, $e_k = (P_k, \textsc{enroll}(k), \mathcal{O}, ack)$.

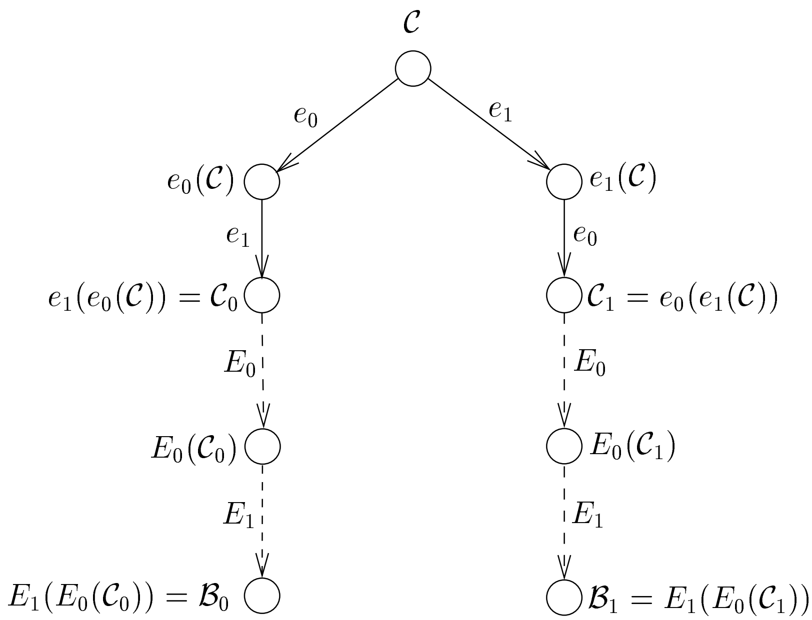


Figure 8: Some configurations reachable from $\mathcal{C}$.

Proof. Immediate from the fact that process P_k has an $\textsc{enroll}(k+2)$ to $\mathcal{O}$ pending in both $\mathcal{B}_0$ and $\mathcal{B}_1$, for each $k \in \{0,1\}$. $\square$

Let S be any schedule of algorithm $\mathbf{A}$ that is applicable to $\mathcal{B}_0$. We say that S is *deciding* if it contains either

- an $\textsc{enroll}(4)$ step to $\mathcal{O}$, or
- an $\textsc{enroll}(5)$ step to $\mathcal{O}$ by a process that appears after a step in S by the other process.

S is *nondeciding* if it is not deciding.

Lemma 14 *Let S be any schedule that is applicable to $\mathcal{B}_0$. If S is nondeciding then*

(a) S is applicable to $\mathcal{B}_1$, and the state of each process and each object except $\mathcal{O}$ is the same in $S(\mathcal{B}_0)$ as in $S(\mathcal{B}_1)$; and

(b) S contains at most one $\textsc{enroll}(2)$ *step to $\mathcal{O}$, at most one* $\textsc{enroll}(3)$ *step to $\mathcal{O}$, and no* $\textsc{enroll}(v)$ *step to $\mathcal{O}$ for any $v \in \{0,1\}$.*

PROOF. (a) By Lemma 12, $\mathcal{B}_0$ and $\mathcal{B}_1$ differ only in that the state of $\mathcal{O}$ in the former is N_{01}, while in the latter it is ☹. As mentioned, both N_{01} and ☹ are white. Since S is nondeciding, S does not contain an ENROLL(4) step to $\mathcal{O}$, or an ENROLL(5) step to $\mathcal{O}$ by a process that appears after a step in S by the other process. Recall that, for each $k \in \{0,1\}$, P_k has an ENROLL$(k+2)$ to $\mathcal{O}$ pending in $\mathcal{B}_0$. Therefore, S does not contain an ENROLL(5) step to $\mathcal{O}$ that appears after both an ENROLL(2) and an ENROLL(3) step to $\mathcal{O}$. Given these facts, it is easy to verify (using Figure 2) that for every prefix S' of S, the state of $\mathcal{O}$ in $S'(\mathcal{B}_0)$ is one of $\{N_{01}, N_{012}, N_{013}, N_{0123}, ☹\}$. Since all these states are white and every operation of type dor is stationary at ☹ (the state of $\mathcal{O}$ in $\mathcal{B}_1$), by Lemma 5, S is applicable to $\mathcal{B}_1$, and the state of each process and each object except $\mathcal{O}$ is the same in $S(\mathcal{B}_0)$ as in $S(\mathcal{B}_1)$.

(b) Suppose, for contradiction, that S contains more than one ENROLL(2) step to $\mathcal{O}$, or more than one ENROLL(3) step to $\mathcal{O}$, or an ENROLL(v) step to $\mathcal{O}$ for some $v \in \{0,1\}$. By inspection of Figure 2, it is easy to verify that, under this assumption, the state of $\mathcal{O}$ in $S(\mathcal{B}_0)$ is ☹. By (a), S is applicable to $\mathcal{B}_1$ and the state of each process and each object except $\mathcal{O}$ is the same in $S(\mathcal{B}_0)$ as in $S(\mathcal{B}_1)$. Since $\mathcal{O}$ is in state ☹ in $\mathcal{B}_1$, it remains in state ☹ in $S(\mathcal{B}_1)$. Thus, the state of each process and object (including $\mathcal{O}$) is the same in $S(\mathcal{B}_0)$ as in $S(\mathcal{B}_1)$. By Lemma 4, $S(\mathcal{B}_0)$ and $S(\mathcal{B}_1)$ have the same valence. This contradicts the fact that $S(\mathcal{B}_0)$ is 0-valent and $S(\mathcal{B}_1)$ is 1-valent. □

The next two lemmata show that the configuration $\mathcal{B}_0$ and the dor object $\mathcal{O}$ satisfy the properties stated at the beginning of this section. That is, starting from $\mathcal{B}_0$, no process can decide unless at least one of them has applied to $\mathcal{O}$ either an ENROLL(4), or an ENROLL(5) *after* the other process has taken at least one step (Lemma 15). Furthermore, if *both* processes are about to execute the ENROLL(4) or ENROLL(5) operation to $\mathcal{O}$ that they must execute before they can decide, then one of the two must execute an ENROLL(4), and the other must execute an ENROLL(5): they can't both be about to execute the same operation (Lemma 16).

Lemma 15 *Let S be any finite schedule that is applicable to $\mathcal{B}_0$ such that some process has decided in $S(\mathcal{B}_0)$. Then S is deciding.*

PROOF. Let P be the process that has decided in $S(\mathcal{B}_0)$. Since $S(\mathcal{B}_0)$ is 0-valent, P decides 0 in $S(\mathcal{B}_0)$. If S is nondeciding, then by Lemma 14(a), S is applicable to $\mathcal{B}_1$, and the state of each process and each object except $\mathcal{O}$ is the same in $S(\mathcal{B}_0)$ as in $S(\mathcal{B}_1)$. In particular, P has the same state in $S(\mathcal{B}_0)$ as in $S(\mathcal{B}_1)$. But then P also decides 0 in $S(\mathcal{B}_1)$. This contradicts the fact that $S(\mathcal{B}_1)$ is 1-valent. □

Lemma 16 *Let S be any schedule applicable to $\mathcal{B}_0$ such that*

- *S is nondeciding;*
- *both processes have taken steps in S; and*
- *for each $k \in \{0,1\}$, process P_k has an* ENROLL(v_k) *to $\mathcal{O}$ pending in $S(\mathcal{B}_0)$ for some v_k.*

Then, $v_0, v_1 \in \{4, 5\}$ *and* $v_0 \neq v_1$.

PROOF. Since S is nondeciding and both processes have taken steps in S, by Lemma 13 and Lemma 14(b), S contains exactly one ENROLL(2) step to $\mathcal{O}$, exactly one ENROLL(3) step to $\mathcal{O}$, and no ENROLL(v) step to $\mathcal{O}$ for any $v \in \{0, 1\}$. Then, by inspection of Figure 2, it is easy to see that $\mathcal{O}$ must be in state N_{0123} in $S(\mathcal{B}_0)$.

For each $k \in \{0, 1\}$, let $c_k = (P_k, \text{ENROLL}(v_k), \mathcal{O}, ack)$. From the specification of dor, each of $\{\, \text{ENROLL}(v) : 0 \leq v \leq 3 \,\}$ is upsetting at N_{0123}. If $v_k \in \{0, 1, 2, 3\}$ for some $k \in \{0, 1\}$, then $\mathcal{O}$ is in state ☹ in $c_k(S(\mathcal{B}_0))$. Since $\mathcal{O}$ is also in state ☹ in $c_k(S(\mathcal{B}_1))$, it follows that each process and each object is in the same state in $c_k(S(\mathcal{B}_0))$ as in $c_k(S(\mathcal{B}_1))$. By Lemma 4, $c_k(S(\mathcal{B}_0))$ and $c_k(S(\mathcal{B}_1))$ have the same valence. This contradicts the fact that $c_k(S(\mathcal{B}_0))$ is 0-valent and $c_k(S(\mathcal{B}_1))$ is 1-valent. Therefore, $v_0, v_1 \in \{4, 5\}$.

To complete the proof it remains to show that $v_0 \neq v_1$. Since c_1 is an ENROLL step to $\mathcal{O}$, it is applicable to both $c_0(S(\mathcal{B}_0))$ and $c_0(S(\mathcal{B}_1))$. Let $A_0 = c_1(c_0(S(\mathcal{B}_0)))$ and $A_1 = c_1(c_0(S(\mathcal{B}_1)))$. If $v_0 = v_1$, then $\mathcal{O}$ is in state ☹ in both A_0 and A_1. Then, each process and each object has the same state in A_1 as in A_0. By Lemma 4, A_0 and A_1 have the same valence. This contradicts the fact that A_0 is 0-valent and A_1 is 1-valent. Hence, it must be that $v_0 \neq v_1$, as desired. □

3.3.3 Stage two of the proof

We now describe an algorithm, derived from **A**, that solves one-resilient Consensus for two processes, Q_0 and Q_1, and uses one dor object less than **A**. The idea is very similar to the algorithm used in [8]: Each process Q_k simulates the actions of the corresponding process P_k in an execution of **A**, and uses this execution to determine what value to decide. In this simulation, the processes are allowed to access all objects used by **A**, except $\mathcal{O}$, each of which is initialised to the state it has in configuration $\mathcal{B}_0$.

Each process Q_k starts its simulation of P_k by pretending that P_k is in the state it has in configuration $\mathcal{B}_0$. Based on the current state of (the simulated) P_k, Q_k determines the operation op that P_k would apply next, and the object O to which op would be applied. If $O \neq \mathcal{O}$, then Q_k has access to O and can apply op to it directly to determine the response and update the (simulated) state of P_k accordingly. If, however, $O = \mathcal{O}$, this is not possible, since Q_k is not allowed to access $\mathcal{O}$. Instead, in this case, Q_k acts as follows:

If op is a REVEAL *operation*, then Q_k pretends that it applied this operation to $\mathcal{O}$ and received response 1.

If op is an ENROLL(v) *operation*, then there are three cases:

CASE 1. $v = k + 2$: Q_k pretends that it applied this operation to $\mathcal{O}$ and received response *ack*.

CASE 2. $v = 4$: Q_k stops simulating steps of P_k in **A** and decides its own initial value.

CASE 3. $v = 5$: Q_k first checks whether $Q_{\overline{k}}$ has simulated its first step of $P_{\overline{k}}$. If not, Q_k pretends that it applied the ENROLL(5) operation to $\mathcal{O}$ and received response *ack*. Otherwise, Q_k stops simulating steps of **A** and decides $Q_{\overline{k}}$'s initial value.

If Q_k does not decide, then it updates the (simulated) state of P_k as if op were applied to $\mathcal{O}$ (as described above) and then proceeds with the simulation of the next operation of P_k.

The algorithm is described more formally in Figure 9. A few remarks about the convention used in the pseudocode are in order. Recall, from Figure 4, that Apply(P_k, op, O) is the procedure by which P_k invokes operation op on object O; it returns the result of this invocation (and updates the state of O accordingly). In addition, we assume that we are given two functions that describe the behaviour of processes P_0 and P_1 in **A** (these correspond to the functions ν and τ discussed when we formally defined processes in Section 2.2).

- NextOp(P_k, s): returns the pair $\langle op, O\rangle$, where the next operation that P_k executes in **A** when in state s is to apply op to object O.
- NextState(P_k, s, r): returns the state that P_k enters in **A** if it receives response r from the operation it invokes when in state s.

In the algorithm, register R_k, $k \in \{0,1\}$, with initial value 0 is used by process Q_k to indicate whether it has simulated a step of P_k, Specifically, Q_k writes 1 into R_k to signify that it has simulated the first step of P_k (lines 14–15). Another shared register D_k is used by Q_k into which it writes its initial value for Consensus before it starts simulating the steps of P_k (line 1). In addition to these shared variables, we also use an auxiliary variable S whose value, informally speaking, is the schedule of steps of **A** that have been simulated by Q_0 and Q_1 so far. This variable is not needed by the algorithm, but it is useful in proving its correctness (Lemma 17). In one atomic step, besides accessing an ordinary variable, each process can also modify the auxiliary variable S. To emphasise this, we bracket with "$[\![\cdots]\!]$" the actions that correspond to an atomic step affecting both an ordinary variable and the auxiliary variable S.

Lemma 17 *If* **A** *is a wait-free Consensus algorithm for processes P_0 and P_1, then the algorithm in Figure 9 is also a wait-free Consensus algorithm for processes Q_0 and Q_1, using one fewer* dor *object.*

PROOF. It is obvious that the algorithm in Figure 9 uses one fewer dor object than **A**. Consider an arbitrary execution of the algorithm in Figure 9. We must prove that in this execution the three properties of Consensus — Termination, Validity and Agreement — are satisfied. First we establish some facts.

Let S_i be the value of the auxiliary variable S when S contains a schedule with i steps. A straightforward induction on all $i \geq 0$ such that S_i is defined shows the following invariants:

(a) S_i is applicable to $\mathcal{B}_0$.

(b) S_i is nondeciding.

(c) Let Q_k be the process that assigns S_i to S, $state_k$ be the value to which Q_k sets its local variable $state$ just after assigning S_i to S, and S_{i+} be the value that Q_k assigns to S next (if any). Then, $state_k$ is the state of process P_k in configuration $S_j(\mathcal{B}_0)$ of **A**, for any j, $i \leq j < i+$.

Shared: all objects used in $\mathbf{A}$ except $\mathcal{O}$, each initialised to the state it has in $\mathcal{B}_0$
D_0, D_1: register, each initialised to $\perp$
R_0, R_1: register, each initialised to 0
S : auxiliary variable : schedule of $\mathbf{A}$ applicable to $\mathcal{B}_0$, initially empty

Code for process Q_k, $k \in \{0, 1\}$

D_k := initial value of Q_k
state := state of P_k in $\mathcal{B}_0$
while Q_k has not decided **do**
    $\langle op, O \rangle$:= NextOp(P_k, state)
    if $O \neq \mathcal{O}$ **then**
        $[\![$ r := Apply(P_k, op, O)
            $S := S \cdot (P_k, op, O, r)$ $]\!]$
            state := NextState(P_k, state, r)
    else (* $O = \mathcal{O}$ *)
            if op = REVEAL **then**
                $S := S \cdot (P_k, \text{REVEAL}, \mathcal{O}, 1)$
                state := NextState(P_k, state, 1)
            else if op = ENROLL($k+2$) **then**
                $[\![$ $R_k := 1$
                $S := S \cdot (P_k, \text{ENROLL}(k+2), \mathcal{O}, ack)$ $]\!]$
                state := NextState(P_k, state, ack)
            else if op = ENROLL(5) **then**
                $[\![$ **if** $R_{\overline{k}} = 0$ **then**
                    $S := S \cdot (P_k, \text{ENROLL}(5), \mathcal{O}, ack)$ $]\!]$
                    state := NextState(P_k, state, ack)
                else
                    decide $D_{\overline{k}}$
            else (* op = ENROLL(4) *)
                decide D_k

Figure 9: Solving wait-free Consensus for two processes using $\mathbf{A}$ with one fewer dor object.

We now turn to the proof that the three properties of a Consensus algorithm are satisfied.

Termination: Suppose, for contradiction, that Q_k is correct and never decides in the execution. This means that the **while** loop of process Q_k does not terminate. Thus, there is an infinite schedule S^* of $\mathbf{A}$ that contains infinitely many steps of P_k such that S^* is applicable to $\mathcal{B}_0$ and is nondeciding (by Invariants (a) and (b) above). By Lemma 15, P_k does not decide in $S'(\mathcal{B}_0)$, for all prefixes S' of S^*. This contradicts the fact that $\mathbf{A}$ is a wait-free Consensus algorithm for two processes (in particular, it contradicts the Termination property of $\mathbf{A}$).

Validity: In any execution of the algorithm in Figure 9, only process Q_k writes into D_k, for each $k \in \{0, 1\}$, and the value Q_k writes there is its initial value (line 1). Thus, to show the execution satisfies Validity, it suffices to show that if Q_k decides the value in D_ℓ, then Q_ℓ has previously written into D_ℓ. There are two cases:

CASE 1. *Q_k decides in line 24:* Then Q_k decides the value in D_k, into which it has previously written.

CASE 2. *Q_k decides in line 22:* Then Q_k reads $R_{\overline{k}} = 1$ in line 18. This means that $Q_{\overline{k}}$ has already executed line 14, and hence line 1 as well. Thus, $Q_{\overline{k}}$ has written into $D_{\overline{k}}$ before Q_k decides the value in $D_{\overline{k}}$.

Agreement: Suppose both Q_0 and Q_1 decide in the execution. (Otherwise, Agreement is trivially satisfied.) Let S^* be the value of S after both processes have decided. (Note that S^* is finite.) For each $k \in \{0, 1\}$, let $state_k$ be the value of the local variable $state$ of Q_k when Q_k decides. By Invariants (a) and (b), S^* is applicable to $\mathcal{B}_0$ and is nondeciding. By Invariant (c), $state_k$ is the state of P_k in $S^*(\mathcal{B}_0)$, and by the algorithm in Figure 9, Q_k decides only when the operation of P_k pending in $S^*(\mathcal{B}_0)$ is $\text{ENROLL}(v_k)$ to $\mathcal{O}$, for some v_k. Also both P_0 and P_1 have taken steps in S^* (because, by the algorithm in Figure 9, each Q_k must simulate at least one step of P_k before deciding). We have thus shown that the hypothesis of Lemma 16 applies for $S = S^*$. Therefore, by Lemma 16, $v_0, v_1 \in \{4, 5\}$ and $v_0 \neq v_1$. Thus, one of Q_0 and Q_1 decides by line 22, while the other decides by line 24. Therefore, both processes decide the value in the same register D_ℓ, for some $\ell \in \{0, 1\}$. By Validity, this happens after Q_ℓ has written into D_ℓ. Since at most one value is written into D_ℓ, both processes decide the value in the same value from D_ℓ. □

Equipped with Lemma 17, we can now prove the main result of this section.

Theorem 2 *There is no wait-free Consensus algorithm for two processes using only* dor *objects and registers.*

PROOF. Suppose, for contradiction, that there is a wait-free Consensus algorithm for two processes using only dor objects and registers. Let $\mathbf{A}$ be such an algorithm that uses a minimal number objects of type dor. Since wait-free Consensus for two processes in unsolvable using registers alone [5, 9], $\mathbf{A}$ uses at least one dor object. By Lemma 17, we can construct a wait-free Consensus algorithm for two processes that uses one fewer dor object, contrary to the definition of $\mathbf{A}$. □

4 The case of four or more processes

In this section we show that for $n \geq 4$, any set of types $\mathcal{S}$ (that includes register) strong enough to solve one-resilient Consensus among n processes is also strong enough to solve one-resilient Consensus among $n-1$ processes. This holds even if $\mathcal{S}$ may contain nondeterministic types. We prove this by showing how a Consensus algorithm for $n \geq 4$ processes can be simulated by just three processes using only registers in addition to the objects already used by the Consensus algorithm. This technique was introduced by Borowsky and Gafni [1] and later used by Chandra *et al.* [3]. Our adaptation of this idea has some novel aspects, however. In Section 4.1 we give a special implementation of type test-and-set-register that is used as a subroutine in our simulation. The simulation itself is described in Section 4.2.

4.1 Type test-and-set-register

The object type test-and-set-register has two states 0 and 1, and supports a single operation TEST&SET. A TEST&SET sets the state to 1 and returns the old state. Thus, the first time TEST&SET is applied to state 0, it changes the state and returns 0; thereafter, any TEST&SET returns 1 leaving the state unchanged. In the sequel we shall refer to an object of type test-and-set-register as a "TAS register".

Our simulation (of a Consensus algorithm for $n \geq 4$ processes by only three processes) uses a non-wait-free implementation of test-and-set-register, initialised to state 0, for three processes. The implementation is not wait-free because some correct process may be unable to finish a TEST&SET operation it has applied if some other process crashes while applying its first TEST&SET operation. However, it guarantees that each crash of a process can prevent at most one correct process from finishing its TEST&SET invocation. More precisely, it ensures that in any execution in which the three processes apply (perhaps repeatedly) TEST&SET operations, the following property holds: The number of correct processes that initiate but cannot finish a TEST&SET invocation does not exceed the number of processes that crash during their first TEST&SET invocation. This property, as we shall see later, is crucial to the correctness of our simulation.

An implementation of TAS registers shared by $s \geq 2$ processes with this property is shown in Figure 10. It is based on a mutual exclusion algorithm discovered independently by Burns and Lynch [2], and Lamport [7]. A shared variable *Closed*, initially false, is used to record whether a TEST&SET has already been invoked on the TAS register. To apply a TEST&SET, a process first checks if *Closed* is true (meaning, that a TEST&SET has been invoked). If so, the process immediately returns 1. Otherwise, the process sets *Closed* to true and tries to enter the critical section (CS) — perhaps competing with other processes that also found *Closed* = false. The TEST&SET operation of the first process to enter the CS returns 0; all others return 1. To determine which process is the first to enter the CS, we use another shared Boolean variable *Done* with initial value false. The first process that enters the CS finds *Done* = false, and sets *Done* to true before it leaves the CS.

Shared: *Active*: **array** $[0..s-1]$ of register, each is a Boolean variable, initially false
Closed: register, a Boolean variable, initially false
Done: register, a Boolean variable, initially false

TEST&SET, return 0 or 1. Code for process Q_k, $k \in \{0, \ldots, s-1\}$

```
1  if Closed then return 1
2  Closed := true
3  Active[k] := true
4  for i := k + 1 to s − 1 do
5      repeat
6          for j := 0 to k − 1 do
7              if Active[j] then
8                  Active[k] := false;  return 1
9      until not Active[i]
   (∗ Beginning of CRITICAL SECTION ∗)
10 if not Done then
11     Done := true
12     result := 0
13 else
14     result := 1
   (∗ End of CRITICAL SECTION ∗)
15 Active[k] := false; return result
```

Figure 10: A non-wait-free implementation of a TAS register using only registers.

Lemma 18 *In any execution of the algorithm in Figure 10, it is impossible that two processes execute the critical section (lines 10–14) at the same time.*

PROOF. Suppose, for contradiction, that two processes, say Q_k and Q_ℓ, are in the critical section (CS) at the same time. Without loss of generality, assume that $k < \ell$. Since Q_ℓ enters the CS, Q_ℓ must find $Active[k] =$ false when it executes line 7 with $j = k$. So, Q_ℓ executes its line 3 before Q_k. (This is because, by our assumption that Q_k and Q_ℓ execute the CS concurrently, after setting $Active[k]$ to true in line 3, Q_k can set $Active[k]$ back to false only in line 15.) Since Q_k also enters the CS, Q_k must find $Active[\ell] =$ false when it executes line 9 with $i = \ell$. So, Q_k executes its line 3 before Q_ℓ — a contradiction. □

Consider an execution E of processes $Q_0, \ldots, Q_{s-1}$ in which each of them invokes (sequentially) zero or more TEST&SET operations by executing the algorithm in Figure 10. We say that Q is *potentially interrupting* in E if it crashes in E during its first TEST&SET invocation, after it has taken at least one step. We say that Q is *blocked* in E if it is correct in E but never returns a response to a TEST&SET invocation it has initiated. Note that a process may be blocked in E only during its *first* TEST&SET invocation (because in all subsequent TEST&SET invocations, the process finds *Closed* = true and returns immediately in line 1). The next lemma shows that the algorithm in Figure 10 implements a TAS register with the "limited blocking" property described at the beginning of this subsection.

Lemma 19 *Consider any infinite execution of processes $Q_0, \ldots, Q_{s-1}$ in which each of them invokes (sequentially) zero or more* TEST&SET *operations by executing the algorithm in Figure 10. In this execution,*

(a) the number of blocked processes does not exceed the number of potentially interrupting processes;

(b) if there is no potentially interrupting process, then some process' first TEST&SET *invocation returns 0; and*

(c) at most one TEST&SET *invocation returns 0.*

PROOF. (a) Consider any blocked process Q_k. By Figure 10 and the definition of a blocked process, it must be that Q_k is correct and executes a nonterminating **repeat** loop in lines 5–9, with $i = k'$, for some $k' > k$. This means that (i) eventually, $Active[k]$ = true forever, and (ii) Q_k finds $Active[k']$ = true (in line 9) infinitely often. By (ii), when Q_k executes for the first time line 9 with $i = k'$, $Q_{k'}$ has already started its first TEST&SET invocation. Also by (ii), Q_k' never returns a value in this invocation. (To see this note that, to return a value, $Q_{k'}$ would do so in line 1, 8 or 15. In each of these cases, after $Q_{k'}$ returns, $Active[k']$ will be false forever, contradicting (ii).) We claim that there is no blocked Q_ℓ with $\ell \in \{k+1, \ldots, k'\}$. If such a process existed, by definition, Q_ℓ would execute a nonterminating **repeat** loop in lines 5–9. By (i), Q_ℓ would eventually find $Active[k]$ = true in line 7 (note that k is in $\{0, \ldots, \ell - 1\}$), and thus Q_ℓ would return 1 instead of being blocked. In particular, this implies that $Q_{k'}$ is potentially interrupting (i.e., $Q_{k'}$ crashes while applying its first TEST&SET operation), and there is no blocked process Q_ℓ such that $\ell \in \{k+1, \ldots, k'-1\}$. Therefore, to each blocked process Q_k, we can associate at most one potentially interrupting process $Q_{k'}$, so that different blocked processes are associated with different potenitally interrupting ones. It follows that the number of blocked processes does not exceed the number of potentially interrupting processes.

(b) Since there is no potentially interrupting process, by (a) every process can finish its first TEST&SET invocation, if there is one (because there can be no blocked processes). Let Q_k be the process with the smallest index that finds *Closed* = false in line 1 during its first TEST&SET invocation. In this invocation, process Q_k does not return in line 8 because it does not execute this line (as it must find $Active[j]$ = false in line 7, for every $j \in \{0, \ldots, k-1\}$). Thus, some process (namely Q_k) finishes its first TEST&SET by returning in line 15. Hence, some process executes the CS during its first TEST&SET invocation. By Lemma 18, there is a well-defined first process to enter the CS. It is easy to see that the first TEST&SET invocation of that process returns 0.

(c) Suppose, for contradiction, that two invocations of TEST&SET, say by processes Q_k and Q_ℓ, both return 0. Thus, in these invocations Q_k and Q_ℓ both execute the CS and find that

Done = false in line 10. By Lemma 18, one of these two processes, say Q_k, finishes the CS before the other enters the CS. Since Q_k finds *Done* = false in line 10, it sets *Done* to true in line 11. Since *Done* is never set to false again, when Q_ℓ later enters the CS it will find *Done* = true, and will return 1, not 0 — contrary to the assumption. □

4.2 The simulation algorithm

Suppose we are given an arbitrary one-resilient Consensus algorithm **A** for n processes, $P_0, P_1, \ldots, P_{n-1}$, for some $n \geq 4$. We make no assumptions about the set of base objects used by this algorithm. We show how, using **A** and only some additional registers, three processes Q_0, Q_1 and Q_2, can solve one-resilient Consensus. Let $\mathcal{Q} = \{Q_0, Q_1, Q_2\}$ and $\mathcal{P} = \{P_0, P_1, \ldots, P_{n-1}\}$.

The main idea is that the processes in $\mathcal{Q}$ will simulate the steps of the processes in $\mathcal{P}$ so that they end up simulating an execution of **A**. As soon as some $P \in \mathcal{P}$ decides in the simulated execution, the processes in $\mathcal{Q}$ will adopt P's decision and solve Consensus. Special care must be exercised to ensure that if at most one process in $\mathcal{Q}$ crashes, the simulated execution of **A** will be one in which at most one process in $\mathcal{P}$ crashes. Since **A** is one-resilient, eventually some process will decide in the simulated execution, and thus so will the correct processes in $\mathcal{Q}$.

To ensure that each step of **A** is simulated, the processes in $\mathcal{Q}$ simulate the steps of processes in $\mathcal{P}$ in round-robin fashion. Specifically, each Q_k tries to simulate the first step of P_0, the first step of P_1 and so on, until the first steps of all processes in $\mathcal{P}$ have been simulated. Then Q_k tries to simulate the second step of P_0, the second step of P_1 and so on, until the second steps of all processes in $\mathcal{P}$ have been simulated.

For the simulation to be proper, however, the processes in $\mathcal{Q}$ must coordinate to ensure that they don't repeat the simulation of a step that has already been simulated. (Such a repetition would be disastrous because then the simulated execution might not be a legitimate execution of **A**. Consider, for example, what would happen if an "increment by one" step is simulated several times!) To prevent this, we use TAS registers shared by Q_0, Q_1 and Q_2. For each i, $0 \leq i \leq n-1$, and each $r \geq 1$, there is a TAS register $TAS[i,r]$ initialised to 0, implemented using the algorithm in Figure 10 (with $s = 3$). Before simulating the rth step of process P_i, process Q_k applies a TEST&SET operation to $TAS[i,r]$. If Q_k wins $TAS[i,r]$ (i.e., Q_k's TEST&SET returns 0), it is the unique process responsible for simulating the rth step of P_i. Otherwise, that step has already been or will be simulated by another process in $\mathcal{Q}$, and Q_k goes on to simulate the step of the next process.

By using $TAS[i,r]$ registers implemented as discussed in Section 4.1, we ensure that the crash of one of the processes in $\mathcal{Q}$, in the middle of applying the TEST&SET operation to $TAS[i,r]$, will block at most one other process' execution of TEST&SET on $TAS[i,r]$ (Lemma 19(a)). Since there are three processes in $\mathcal{Q}$, there will still remain a process in $\mathcal{Q}$ which neither crashes nor is blocked by the crash. This process will continue simulating steps of the processes in $\mathcal{P}$ (except for P_i — the process in the simulation of whose step the crash occurred) until one of them decides. (This, incidentally, is the reason why we need at least three simulator processes, and the reason why this simulation cannot be made to work in the case of $n = 3$ — discussed in Section 3.)

The simulation algorithm is shown in Figure 11. In addition to the $TAS[i, r]$ registers already mentioned, the simulation uses the following variables:

- *Instr*: an array $[0..n-1]$ of n registers. $Instr[i]$ contains the index of the first instruction of P_i that has not been simulated yet.
- *State*: an array $[0..n-1]$ of n registers. $State[i]$ contains the current (simulated) state of P_i.
- *Decision*: a register. If a process Q in $\mathcal{Q}$ simulates a step of some process P in $\mathcal{P}$ and as a result of which P decides, then Q writes P's decision into *Decision*.

Besides the above shared variables, the simulation also uses two auxiliary variables, S and W. S is the schedule of **A** consisting of the steps simulated so far. W is a two dimensional array that keeps track of which process in $\mathcal{Q}$ has simulated which step of each process in $\mathcal{P}$. Specifically, if the rth step of P_i has been simulated by Q_k, then $W[i, r] = k$; if the rth step of P_i has not been simulated yet, then $W[i, r] = \perp$.

The simulation of **A** by Q_0, Q_1 and Q_2 proceeds as follows. Each Q_k enters a **while** loop (lines 2–20), from which it will exit when a process in $\mathcal{P}$ has decided in the simulated execution of **A**. In each iteration of the loop, Q_k will attempt to simulate the "next" step of process P_i, which is step $r = Instr[i]$ (line 3). To determine if Q_k should simulate this step, Q_k applies a TEST&SET to $TAS[i, r]$. (For now, ignore the **coexecute** statement in lines 5–8, and replace it by line 6. We shall explain the meaning of and need for the **coexecute** statement shortly.) If Q_k wins $TAS[i, r]$ (line 9), it simulates the next step of P_i (line 14). If this is the first step of P_i, i.e., $r = 1$, Q_k uses its own initial value as the initial value of P_i (lines 10–12). Next, Q_k updates the state of the simulated process in $State[i]$ (line 15). If the simulated step causes P_i to decide, then that decision is written into *Decision* (line 17). Otherwise, $Instr[i]$ is incremented by one to reflect the fact that another step of P_i has been simulated (line 19), and Q_k turns its attention to the next process in round-robin order (line 20).

In addition to the notation and conventions introduced in conjunction with the simulation described in Figure 9, we also use the construct "**coexecute** M_1 ▮ M_2 **coend**", where M_1 and M_2 are arbitrary statements. This interleaves the execution of steps of the two (potentially nonterminating) statements, until one of the two terminates. When (and if) that occurs, the execution of steps of the other statement is abandoned, and the **coexecute** statement itself terminates.

We now explain the reason for using the **coexecute** statement (lines 5–8). Consider an execution of the simulation in Figure 11 in which a process in $\mathcal{Q}$, say Q_k, crashes while it is executing its first TEST&SET on $TAS[i, r]$. Suppose that the other two processes are correct in this execution. By Lemma 19(a), the crash of Q_k may cause at most one correct process, say Q_ℓ, to be blocked in an TEST&SET invocation on $TAS[i, r]$. Therefore, the remaining correct process in $\mathcal{Q}$, say Q_m, will not be blocked in any of its TEST&SET invocations on $TAS[i, r]$. Thus, Q_m will be able to simulate steps of processes in $\mathcal{P}$ (other than P_i) until

```
Shared:  all objects used in A, each has the initial state specified by A
         TAS: array [0..n − 1, 1..∞] of test-and-set-register,
                  each implemented by the algorithm in Figure 10
         Instr: array [0..n − 1] of register, each initialised to 1
         State: array [0..n − 1] of register, each initialised to ⊥
         Decision: register, initialised to ⊥
         S: auxiliary variable, containing a schedule of A, initially empty
         W: auxiliary variable, contains an array [0..n − 1, 1..∞] of {⊥, 0, 1, 2},
                  each initially ⊥

Code for process Q_k, k ∈ {0, 1, 2}
id := 0
while Decision = ⊥ do
    rd := Instr[id]
    winner := 1
    coexecute
        winner := Apply(Q_k, TEST&SET, TAS[id, rd])
        ▮
        repeat until Decision ≠ ⊥
    coend
    if winner = 0 then
        (* Q_k wins TAS[id, rd] and must simulate the rd th step of P_id *)
        if rd = 1 then   (* first step of P_id *)
            let u be the initial value of process Q_k
            State[id] := initial state of P_id where P_id has initial value u
        ⟨op, O⟩ := NextOp(P_id, State[id])
       ⟦ res := Apply(P_id, op, O);  W[id, rd] := k;  S := S · (P_id, op, O, res) ⟧
        State[id] := NextState(P_id, State[id], res)
        if P_id has decided in State[id] then
            Decision := the value decided by P_id
        else
            Instr[id] := Instr[id] + 1
    id := (id + 1) mod n
decide Decision
```

Figure 11: Solving one-resilient Consensus for three processes using **A**.

one of them decides. Q_m will then write its decision into *Decision* and after breaking out of the **while** loop (lines 2–20), it will decide. Remember, however, that there is also a correct process in $\mathcal{Q}$, namely Q_ℓ, that is blocked in a TEST&SET invocation on $TAS[i,r]$. By applying TEST&SET to $TAS[i,r]$ as one branch of a **coexecute** statement (line 6) and checking whether $Decision = \perp$ in another branch (line 7), we ensure that Q_ℓ will eventually break out of the potentially nonterminating TEST&SET invocation, and will decide.

Lemma 20 *If* **A** *is a one-resilient Consensus algorithm for $n \geq 4$ processes $P_0, \ldots, P_{n-1}$ using $\mathcal{S}$, then the algorithm in Figure 11 is a one-resilient Consensus algorithm for three processes Q_0, Q_1, Q_2 using $\mathcal{S}$.*

PROOF. By inspection, besides the shared objects used in **A**, all the additional nonauxiliary shared objects used by the algorithm in Figure 11 are registers. Since $\mathcal{S}$ contains type register, it follows that the algorithm uses only objects of types in $\mathcal{S}$.

We now show that the algorithm in Figure 11 solves one-resilient Consensus for the three processes in $\mathcal{Q}$. To this end, in the rest the proof we fix an arbitrary execution of the algorithm in Figure 11 in which at most one process in $\mathcal{Q}$ crashes. Henceforth, our discussion is made with respect to this fixed execution. We shall prove that in this execution the three properties of Consensus — Termination, Validity and Agreement — are satisfied.

First we make some definitions. Let S_j be the value of S when S contains a schedule of j steps (undefined if S never has j steps). Let W_j be the value assigned to W when S_j is assigned to S (both auxiliary variables are updated in the same "atomic" action in line 14.) Note that if $W_j[i,r] \neq \perp$, then for all $j' \geq j$ where $W_{j'}$ is defined, $W_{j'}[i,r] = W_j[i,r]$. This is because at most one process in $\mathcal{Q}$ wins $TAS[i,r]$ (Lemma 19(c)), and that is the only process that may assign its id to $W[i,r]$. Let $\mathcal{I}_j$ be the set of initial configurations I of **A** that satisfy the following property: If $W_j[i,1] = k \neq \perp$, then the initial value of P_i in I is equal to the initial value of Q_k. That is, all the initial configurations in $\mathcal{I}_j$ agree on exactly the initial values of all processes in $\mathcal{P}$ that have at least one step in S_j; the initial value of any such process, say P_i, is equal to the initial value of the process in $\mathcal{Q}$ that simulated P_i's first step in S_j (i.e., the process Q_k such that $W_j[i,1] = k$). Note that if $j \leq j'$, then $\mathcal{I}_j \supseteq \mathcal{I}_{j'}$.

We now define S^* and W^*. The intuition behind these definitions is that S^* and W^* are the final values of the auxiliary variables S and W, if these variables eventually stop being updated. If S and W keep changing forever, S^* and W^* are the "limit" values of these variables. More precisely, S^* is defined as follows: if S_j is defined then the jth step of S^* is the last step of S_j; otherwise, S^* does not have a jth step. W^* is defined as follows:

$$W^*[i,r] = \begin{cases} \perp & \text{if for every } j,\ W_j[i,r] = \perp, \\ k & \text{if for some } j,\ W_j[i,r] = k \neq \perp. \end{cases}$$

Note that W^* is well-defined because if $W_j[i,r] = k$ then, for any $j' \geq j$ such that $W_{j'}$ is defined, $W_{j'}[i,r] = k$ as well. Finally, we define $\mathcal{I}^*$ (derived from W^*, just as $\mathcal{I}_j$ is derived from W_j) as the set of initial configurations I of **A** that satisfy the following property: If $W^*[i,1] = k \neq \perp$, then the initial value of P_i in I is equal to that of Q_k.

With these definitions, it is easy to show that the following properties are invariants of the algorithm in Figure 11: For all $j \geq 0$ such that S_j is defined,

(a) S_j is a schedule of $\mathbf{A}$ that is applicable to every $I \in \mathcal{I}_j$.

(b) $W_j[i, r] \neq \perp$ if and only if S_j has at least r steps of P_i.

(c) Let P_i be the process such that the last step of S_j is a step of P_i; let σ be the value assigned to $State[i]$ just after S_j is assigned to S; and let $j+$ be the minimum integer greater than j such that S_{j+} is defined and the last step of S_{j+} is a step of P_i, or ∞ if no such $j+$ exists (i.e., if the last step of P_i ever assigned to S is the last step of S_j). Then, for every k, where $j \leq k < j+$, and every $I \in \mathcal{I}_k$, σ is the state of P_i in $S_k(I)$.

We are now ready to prove that the fixed execution satisfies the three properties of Consensus. For each $i \in \{0, \ldots, n-1\}$ and $r \geq 1$, let $E_{i,r}$ denote the subexecution consisting of only steps taken the processes when they apply TEST&SET operations to $TAS[i, r]$ in line 6.

Termination: Assume, for contradiction, that some correct process never decides. Then, that process must always find $Decision = \perp$ in line 2 or line 7. Therefore, no process (correct or crashed) ever executes line 17 — because once $Decision$ is set to a non-$\perp$ value, it is never set back to $\perp$. Hence,

$$Decision = \perp, \text{ forever.} \tag{$*$}$$

Claim 2 *At least one correct process executes the* **while** *loop (lines 3–20) infinitely often.*

PROOF. Suppose, for contradiction, that no correct process executes the **while** loop (lines 3–20) infinitely often. Since at most one process in $\mathcal{Q}$ may crash, there are at least two correct processes, say Q and Q'. By assumption, neither Q nor Q' executes the **while** loop infinitely often. By line 2 of Figure 11 and ($*$), these two processes cannot exit the **while** loop. This means that eventually each of Q and Q' applies a nonterminating **coexecute** statement (lines 5–8). Let i and r be the values of Q's local variables id and rd after Q has started its nonterminating **coexecute** statement; let i' and r' be values with corresponding interpretations for Q'. By line 6, it must be that Q and Q' apply a nonterminating TEST&SET operation to $TAS[i, r]$, and $TAS[i', r']$, respectively. In other words, Q and Q' are blocked in $E_{i,r}$ and $E_{i',r'}$, respectively. By Lemma 19(a), there must exist two distinct processes such that one is potentially interrupting in $E_{i,r}$ while the other is potentially interrupting in $E_{i',r'}$. But then there are two processes that have crashed (while applying their first TEST&SET operation to $TAS[i, r]$ and $TAS[i', r']$, respectively). This contradicts our assumption that at most one process in $\mathcal{Q}$ may crash. ☐ Claim 2

Claim 3 *For any $i \in \{0, 1, \ldots, n-1\}$, if $Instr[i]$ is bounded with maximum value r, then some process in $\mathcal{Q}$ crashes either (a) during its first invocation of* TEST&SET *on $TAS[i, r]$ or (b) after winning $TAS[i, r]$ in line 6 but before incrementing $Instr[i]$ in line 19.*

PROOF. Suppose that $Instr[i]$ is bounded and attains maximum value r. By Claim 2, some correct process, say Q_k, executes the **while** loop (lines 3–20) infinitely often. Since in each iteration its local variable id is incremented by one modulo n (line 20), Q_k executes the

while loop with $id = i$ infinitely often. Since, from a certain point on, $Instr[i] = r$ forever, it follows that Q_k invokes infinitely many TEST&SET operations on $TAS[i, r]$. Q_k must lose (i.e., return 1 in) all of these TEST&SET operations: Otherwise, it would increment $Instr[i]$ to $r + 1$ in line 19, contradicting that r is the maximum value that $Instr[i]$ attains.

If no process in $\mathcal{Q}$ crashes while executing a TEST&SET operation on $TAS[i, r]$ (i.e., if possibility (a) of the claim does not hold), then no process is potentially interrupting in $E_{i,r}$. Recall that $E_{i,r}$ is not empty, because Q_k invokes infinitely many (losing) TEST&SET operations on $TAS[i, r]$. Thus, by Lemma 19(b), some process in $\mathcal{Q}$ wins $TAS[i, r]$. If that process does not crash before incrementing $Instr[i]$ to $r + 1$ in line 19 (i.e., if possibility (b) of the claim also does not hold) then we again contradict the assumption that r is the maximum value that $Instr[i]$ attains. It follows that either (a) or (b) must hold. □ Claim 3

By Claim 3, there is at most one element $i \in \{0, 1, \ldots, n-1\}$ such that $Instr[i]$ is bounded. (Otherwise, there would have to be two processes in $\mathcal{Q}$ that crash.) If there is such an element, let $\hat{\imath}$ be it — otherwise, let $\hat{\imath}$ be an arbitrary element of $\{0, 1, \ldots, n-1\}$. Thus, for all $i \in \{0, \ldots, n-1\} \setminus \{\hat{\imath}\}$ the value of $Instr[i]$ is unbounded. Before incrementing $Instr[i]$ to $r+1$ (in line 19), a process in $\mathcal{Q}$ must have previously set $W[i, r]$ to a non-$\perp$ value (line 14). Thus, for all $i \in \{0, \ldots, n-1\} \setminus \{\hat{\imath}\}$ and all $r \geq 1$, $W^*[i, r] \neq \perp$. By Invariant (b), S^* contains infinitely many steps of P_i, for all $i \in \{0, \ldots, n-1\} \setminus \{\hat{\imath}\}$. This means that S^* is an infinite schedule in which at most one process in $\mathcal{P}$ crashes. By Invariant (a), S^* is a schedule of **A** that is applicable to every $I \in \mathcal{I}^*$. By $(*)$, for every $I \in \mathcal{I}^*$ and $i \neq \hat{\imath}$, there is no prefix S' of S^* such that P_i has decided in $S'(I)$. This contradicts the fact that **A** is a one-resilient Consensus algorithm for n processes (in particular, it contradicts the Termination property of **A**).

We reached this contradiction by assuming that the Termination property in violated. Thus, Termination is satisfied.

Agreement and Validity: By the Termination property shown above, S^* is a finite schedule.

Claim 4 *For any process $Q \in \mathcal{Q}$ and value $v \in \{0, 1\}$, if Q writes v into Decision (line 17), then some process in $\mathcal{P}$ decides v in $S^*(I)$, for every $I \in \mathcal{I}^*$.*

PROOF. Let i and r be the values of its local variables id and rd when Q writes v into *Decision* in line 17. Let σ be the value that Q last wrote into $State[i]$ in line 15 before Q assigns v to *Decision*. In other words, Q simulates the rth step of P_i; as a result of this step P_i enters state σ in which it decides v (cf. line 16).

We first show that σ is the last value assigned to $State[i]$. Suppose, for contradiction, that some process in $\mathcal{Q}$ assigns a value to $State[i]$ after Q assigned σ to it. Thus, some process in $\mathcal{Q}$ simulates step $r + 1$ of P_i. For this to happen, some process $Q' \in \mathcal{Q}$ must assign $r + 1$ to $Instr[i]$. From the algorithm it is clear that the only process that may do so is the process that simulates the rth step of P_i, i.e., the process that wins $TAS[i, r]$. By Lemma 19(c), only one process can win $TAS[i, r]$ and by assumption that process is Q (because Q simulates the rth step of P_i). Thus, Q assigns $r + 1$ to $Instr[i]$ in line 19. By assumption, however, after winning $TAS[i, r]$, Q executes line 17, and therefore does not execute line 19. This contradiction implies that σ is the last value assigned to $State[i]$.

Since σ is the last value assigned to $State[i]$, by Invariant (c), σ is the state of P_i in $S^*(I)$, for every $I \in \mathcal{I}^*$. Since Q assigns v to *Decision* in line 17, it must be that P_i decides v in σ. Thus, P_i decides v in $S^*(I)$, for every $I \in \mathcal{I}^*$ — as wanted. $\square$ Claim 4

To prove that Agreement is satisfied, suppose that two distinct processes in $\mathcal{Q}$ decide the values v and v', respectively, that they read from *Decision*. Thus, both v and v' are written into *Decision* by some processes in $\mathcal{Q}$ in line 17. By Claim 4, some process in $\mathcal{P}$ decides v in $S^*(I)$ and some process in $\mathcal{P}$ decides v' in $S^*(I)$, for every $I \in \mathcal{I}^*$. By the Agreement property of $\mathbf{A}$, $v = v'$. Therefore, the algorithm in Figure 11 satisfies Agreement.

Finally we prove that Validity is satisfied. This is obvious if both 0 and 1 are initial values of the processes in $\mathcal{Q}$. Thus, suppose that all processes in $\mathcal{Q}$ have initial value v, for some $v \in \{0, 1\}$. Let I_v be the initial configuration of $\mathbf{A}$ in which every process in $\mathcal{P}$ has initial value v. Clearly, $I_v \in \mathcal{I}^*$. Suppose now that some process in $\mathcal{Q}$ decides the value u that it reads from *Decision*. By Claim 4, some process P in $\mathcal{P}$ decides u in $S^*(I)$, for every $I \in \mathcal{I}^*$. In particular, P decides u in $S^*(I_v)$. By the Validity property of $\mathbf{A}$, $u = v$. Thus, if all processes in $\mathcal{Q}$ have initial value v, then any process that decides, must decide v. Therefore, the algorithm in Figure 11 satisfies Validity. $\square$

Theorem 3 *Let $\mathcal{S}$ be any set of object types that includes* register. *For any $n \geq 4$, if $\mathcal{S}$ implements one-resilient Consensus among n processes, then $\mathcal{S}$ implements one-resilient Consensus among three (and therefore among $n-1$) processes.*

PROOF. Let $\mathbf{A}$ be a one-resilient Consensus algorithm for $n+1$ processes, where $n \geq 3$, that uses only objects of types in $\mathcal{S}$. By Lemma 20, we can construct a one-resilient Consensus algorithm for three processes that also uses only objects of types in $\mathcal{S}$. Thus, one-resilient Consensus among three processes is solvable using $\mathcal{S}$. $\square$

The simulation in Figure 11 uses an unbounded number of additional registers (because it uses an unbounded number of TAS registers, each of which requires a fixed number of registers). In the remainder of this subsection, we briefly describe how to rectify this shortcoming. Roughly speaking, we shall modify the algorithm in Figure 10 so that it implements a TAS register with an additional operation, called RESET, using only a bounded number of registers. A RESET operation applied to a TAS register causes the register to enter state 0 and returns *ack*. For the purposes of our simulation, we may assume that a RESET operation can be applied only when the TAS register is in state 1, and that no two processes can apply RESET concurrently. Here are the required modifications.

The new algorithm uses *four* sets of shared registers, $\{\, Active_i, Closed_i, Done_i : 1 \leq i \leq 4 \,\}$, used by the algorithm in Figure 10. Also, it maintains a shared register, *Pointer*, and a shared **array** $[0..2]$ of three registers, *Current*; each of these registers can hold an integer in $\{1, \ldots, 4\}$ and is initialised to 1. Using these shared registers, the TEST&SET and RESET operations are implemented as follows. To apply a TEST&SET operation, process Q_k first assigns the value in *Pointer* to *Current*$[k]$. Then, Q_k proceeds as specified in Figure 10, using the shared variables $Active_i$, $Closed_i$ and $Done_i$, where i is the value assigned to *Current*$[k]$ at the beginning of the TEST&SET operation. To apply a RESET, process Q_k first finds an integer i in $\{1, \ldots, 4\}$ that does not appear in array *Current*. (Such an integer must exist because there are at most three distinct integers in *Current*.) Next, Q_k assigns false to every entry in $Active_i$ and to $Closed_i$ and $Done_i$. Finally, Q_k completes the RESET operation by writing i into *Pointer*.

Using the "resettable" TAS registers as described in the preceding paragraph, we now show how to modify the algorithm in Figure 11 so that it uses only a bounded number of registers. Let $RTAS[0..n-1]$ be an array of n resettable TAS registers, and let $First[0..n-1]$ be an array of n Boolean registers, each initialised to true. Here are the required changes to the algorithm in Figure 11.

- Delete line 3 (the shared array *Instr* is no longer necessary).
- Replace the "nonresettable" TAS register $TAS[id, rd]$ in line 6 with the resettable one $RTAS[id]$ (thus the unbounded array *TAS* is replaced by the bounded array *RTAS*).
- Replace "$rd = 1$" in line 10 with "$First[id]$, and insert an assignment statement, "$First[id] :=$ false", between line 10 and line 11.
- Replace the instruction in line 19 by "Apply(Q_k, RESET, $RTAS[id]$)".

By the specification of resettable TAS registers, for all $i \in \{0, \ldots, n-1\}$, at most one process in $\mathcal{Q}$ can win $RTAS[i]$, after each RESET operation is applied to it. Therefore, at most one process in $\mathcal{Q}$ can simulation the next step of P_i at any given time. Given this fact, it is straightforward to verify that the above modifications do not affect the correctness of the simulation. Thus, the resulting simulation algorithm uses only bounded number of additional shared registers, as desired.

5 Conclusion

The results of this paper, together with those in [3], completely characterise the relationship between the solvability of Consensus among different numbers of processes. More precisely, the results show that, given any set $\mathcal{S}$ of object types, the statement

> t-resilient Consensus among n processes is solvable using $\mathcal{S}$ *iff*
> t-resilient Consensus among $n-1$ processes is solvable using $\mathcal{S}$

is valid if and only if $t \geq 2$ or $n \geq 4$. Besides this characterisation, these results also reveal a qualitative difference between level one and other levels of the Consensus hierarchy [5, 6]. If $\mathcal{S}$ contains an object type at level two or above of the Consensus hierarchy, the result of Chandra *et al.* implies that the statement above is valid. On the other hand, if $\mathcal{S}$ contains only object types at level one of the Consensus hierarchy, our result in Section 3 shows that the statement is false. This shows that level one of the Consensus hierarchy may have a richer structure than other levels.

Type dor defined in Section 3 is the only deterministic type we know of that is at level one of the Consensus hierarchy and has no one-resilient implementation for three or more processes using only registers.

The simulation sketched in Section 4.2 was first used by Borowsky and Gafni [1], and later by Chandra *et al.* [3]. The main innovation here is the TAS register implementation of Section 4.1. Borowsky and Gafni used a simulation that requires processes to agree on the outcome of each step by solving (a restricted form of) Consensus using only (read/write) registers. Instead of agreeing on the outcome of a step, we use (a similarly restricted form of) TAS registers to ensure that only one process simulates each step. The simulation of Chandra *et al.* applies to a context in which TAS registers are available directly and need not be implemented.

Acknowledgements

Our thinking on shared memory distributed computing has been deeply influenced from discussions with Tushar Chandra, Prasad Jayanti and Sam Toueg. We are grateful to Faith Fich for her helpful comments on earlier drafts of this paper.

References

[1] Elizabeth Borowsky and Eli Gafni. Generalized FLP impossibility result for t-resilient asynchronous computations. In *Proceedings of the Twenty-Fifth ACM Symposium on Theory of Computing*, pages 91–100, May 1993.

[2] James Burns and Nancy Lynch. Mutual exclusion using indivisible reads and writes. In *Proceedings of Eighteenth Annual Allerton Conference on Communications, Control and Computing*, pages 833–842, 1980.

[3] Tushar Chandra, Vassos Hadzilacos, Prasad Jayanti, and Sam Toueg. Wait-freedom versus t-resiliency and the robustness of wait-free hierarchies. In *Proceedings of the Thirteenth ACM Symposium on Principles of Distributed Computing*, pages 334–343, August 1994.

[4] Michael J. Fischer, Nancy A. Lynch, and Michael S. Paterson. Impossibility of distributed consensus with one faulty process. *Journal of the ACM*, 32(2):374–382, April 1985.

[5] Maurice Herlihy. Wait-free synchronization. *ACM Transactions on Programming Languages and Systems*, 11(1):124–149, January 1991.

[6] Prasad Jayanti. On the robustness of Herlihy's hierarchy. In *Proceedings of the Twelfth ACM Symposium on Principles of Distributed Computing*, pages 145–158, August 1993.

[7] Leslie Lamport. The mutual exclusion problem: Parts I & II. *Journal of the ACM*, 33(2):313–348, 1986.

[8] Wai-Kau Lo. More on t-resilence vs. wait-freedom. In *Proceedings of the Fourteenth ACM Symposium on Principles of Distributed Computing*, pages 110–119, August 1995.

[9] Michael Loui and Hosame Abu-Amara. Memory requirements for agreement among unreliable asynchronous processes. In *Advances in Computer Research*, volume 4, pages 163–183. JAI Press Inc., 1987.

[10] Michael Stumm and Songnian Zhou. Fault tolerant distributed shared memory. In *Proceedings of the Second IEEE Symposium on Parallel and Distributed Processing*, pages 719–724, December 1990.