All of us are smarter than any of us: more on the robustness of the Consensus hierarchy (Part I)

Wai-Kau Lo* Vassos Hadzilacos†

wklo@cs.utoronto.ca vassos@cs.utoronto.ca

Technical Report CSRI-348

June 27, 1996

Abstract

We consider objects accessed by concurrent processes. The Consensus hierarchy [7, 9] classifies object types on the basis of their strength in supporting wait-free implementations of other types. In the context of the present paper, an implementation may use any number of objects of the given types, as well as read/write registers. We prove that, if nondeterministic types are allowed, the Consensus hierarchy is not robust: It is possible to implement objects of types classified as "strong" by combining objects of types that are individually classified as "weak". Thus, in general, it is not possible to determine the power of a concurrent system that supports a given set of primitive object types by reasoning about the power of each primitive type in isolation.

1 Introduction

1.1 Background and overview of the result

Interest in asynchronous shared-memory distributed computing dates back at least to Dijkstra's work on mutual exclusion [3]. In this model of computation, a set of processes cooperate towards some common task (such as excluding each other from a critical section) by asynchronously taking steps, some of which access shared registers. The operations that the processes can apply to the shared registers may be simple read and write operations, or more complex ones such as test-and-set, fetch-and-add, and so forth.

*Supported by a Canadian Commonwealth Scholarship.

†Supported by a grant from the Natural Sciences and Engineering Research Council of Canada.

In a more general version of this model of computation, influenced by the advent of the object-oriented paradigm, processes communicate by accessing shared objects. An object is accessed by applying an operation to it; the operation changes the state of the object and returns a value to the process that invoked it. Each object belongs to a type. A type consists of a set of states, a set of operations, a set of responses to operations, and a state transition relation that describes the effect of applying each operation to an object of that type. More specifically, the state transition relation specifies the responses the object may return and the states it may enter if an operation is applied while the object is in a particular state. Examples of types include register (with read and write operations only), test-and-set-register (with test-and-set operation), stack, queue and priority-queue, among many others. (We use the sans serif font to indicate object types.)

A distributed system supports, in hardware or software, a certain fixed set of "base" object types. To enhance the usability of such a system, a programmer may wish to enrich the set of available object types by implementing new types, not directly supported by the system. To implement a type T from a set of base types $\mathcal{B}$ in a system of n processes, the programmer must provide n procedures for each operation op of T; these are the procedures that may be invoked by the n processes to apply op to an object of type T. All the shared objects used in the procedures written by the programmer must belong to types in $\mathcal{B}$. We assume that the set of base types includes register. This is a reasonable assumption, as it is hard to imagine a shared-memory system that does not provide at least this much.[1] For convenience, in the rest of the paper we say simply "register" instead of "object of type register".

An implementation is required to be *linearisable* [8]: Any concurrent execution E consisting of invocations of the procedures written by the programmer must be equivalent to a *sequential* execution E_s consisting of the same procedure invocations, where each invocation returns a response and changes the state of the object it accesses in accordance with the state transition relation of the implemented type. Moreover, the sequential execution E_s must respect the order of procedure invocations that did not overlap in E; it may serialise in any way procedure invocations that were active concurrently in E.

Of special interest are *wait-free* implementations. In such implementations, a process is guaranteed to complete any procedure it invokes within a finite number of its own steps, regardless of whether other processes are fast, slow, or even crashed (i.e., halted before completing). The requirement of wait-freedom precludes the use of critical sections and busy waiting. In this paper, we consider only wait-free and linearisable implementations.

A fundamental question is whether, in a system of n processes, it is possible to implement a type T from a given set of base types $\mathcal{B}$. In a seminal paper [7], Herlihy established an important link between a type's ability to support implementations of other types and its ability to solve the well-known Consensus problem [5]. More specifically, he showed that, if objects of type T (together with registers) can be used to solve Consensus among

[1] More restricted notions of implementation have appeared in the literature. For instance, in some definitions registers are not taken for granted, while in others only one object of each base type may be used in the implementation. For further discussion on such more restricted definitions, see [9] and [10].

n processes, then T can be used to implement *any* type in a system of n processes. We refer to this fact as "the universality of Consensus".

Define the *Consensus number* of a type T, as the maximum number n such that there is an algorithm that solves Consensus among n processes using only objects of type T (and registers) — or ∞, if no such maximum exists. We can now place each type in the *Consensus hierarchy* according to its Consensus number. By the universality of Consensus, the relative strength of two types T and T' is captured by their levels, say n and n' respectively, in the Consensus hierarchy. If $n \geq n'$, then T is at least as strong as T': in a system of n processes, T can be used to implement T'. If, further, $n \neq n'$, then T is strictly stronger than T': T' cannot be used to implement T in a system of n processes.

Although Consensus numbers allow us to make *pairwise* comparisons on the strength of types, it is not clear whether they can also be used to answer the more general implementability question: Can we implement T from a *set* of base types $\mathcal{B}$, in a system of n processes? Jayanti [9] observed that Consensus numbers could be used to answer this question if the Consensus hierarchy has the property of *robustness*: i.e., no type at level n can be implemented from types at levels strictly below n, in a system of n processes.

Borowsky *et al.* [1], and Peterson *et al.* [15] proved that if we restrict our attention to *deterministic* types, the Consensus hierarchy is robust. A type is deterministic, if whenever we apply an operation to the type, the current state uniquely determines the response of the operation and the new state of the type.

In this paper we prove that if we allow nondeterministic types, then the Consensus hierarchy is *not* robust. Specifically, we demonstrate two nondeterministic (but finite) object types, negation and booster; each of these is at level one of the Consensus hierarchy, but, when combined, they can implement any type in a system of two processes. The central idea behind our proof is the formulation of *Equality Negation*, a weaker variant of the Consensus problem. In Equality Negation, each of two processes starts with an initial value drawn from the set $\{0, 1, 2\}$, and must eventually decide irrevocably either 0 or 1. The two processes must decide the same value if and only if their initial values were distinct. An object of type negation enables two processes to directly solve Equality Negation. An object of type booster allows two processes to solve Consensus provided they can prove that they can solve Equality Negation between them using initial values specified by the booster object.

We give a simple algorithm that allows two processes to solve Consensus using a negation object, a booster object and registers. (This algorithm suggests itself almost immediately from the definition of the booster type.) By the universality of Consensus, this implies that the two types together can implement any type in a system of two processes. The difficult part of our result is in proving that each of negation and booster has Consensus number one. The usual technique for proving upper bounds on the Consensus number of a type, known as the "bivalence argument", works well for deterministic types but runs into difficulties in the context of nondeterministic types. The bulk of the technical work in the paper (Sections 7 and 8) seeks ways to circumvent these difficulties.

1.2 Related work

Jayanti was the first to raise the question of robustness of the Consensus hierarchy [9]. Before his work, it had been widely, though tacitly, assumed that the hierarchy is robust. Jayanti showed that various natural variants of the Consensus hierarchy (including the one used in [7]) are not robust. He also proved that, if a robust hierarchy that classifies types according to their ability to implement other types exists, it must be either the Consensus hierarchy or a coarsening of it — i.e., a hierarchy obtained by collapsing together adjacent levels of the Consensus hierarchy.

As mentioned earlier, Borowsky *et al.* [1] and Peterson *et al.* [15] proved that the Consensus hierarchy, restricted to deterministic types, is robust.

In the next paragraph we describe some results concerning the (non)robustness of the Consensus hierarchy in a model of objects that differs from the "standard" model introduced in [7] (and used in the present paper). To explain the different model recall that, in a system of n processes, each object of type T has n procedures for each operation op of T. Processes trying to apply op to an object concurrently must invoke different procedures. In the "standard" model of objects a process that wishes to apply op to the object invokes one of these n procedures, and remains bound to it only for the duration of the invocation. Other processes may later use that procedure. In the alternative model, a process binds itself to a procedure permanently. Objects in this model are sometimes called *hardwired*, since the binding of processes to procedures is fixed.

Chandra *et al.* [2] proved that, in the model of hardwired objects, the Consensus hierarchy is not robust if nondeterministic types are allowed. Moran and Rappoport [14] strengthen this result by proving that it applies even if only deterministic types are allowed. We borrowed from [2] the idea of a booster type, which "boosts" the power of a weak type T, if it is satisfied that processes have access to T. The booster of [2] promotes a type that solves Consensus between two processes to a type that solves Consensus between three processes. In our case, the booster promotes a type that solves Equality Negation between two processes to a type that solves Consensus between two processes.

Closest to our work is a recent paper of Schenk [16]. He also exhibits two nondeterministic object types each of which (together with registers) is individually too weak to solve Consensus between two processes, but which taken together can solve Consensus for any number of processes. Our result strengthens this in two related ways: First, Schenk's result requires a type that exhibits infinite nondeterminism, while our types exhibit only bounded nondeterminism. Second, and perhaps more importantly, his result requires a definition of wait-freedom that is stronger than the usual definition in the context of types with infinite nondeterminism. We explain these differences in more detail below.

A type has *infinite* nondeterminism if there is a state from which there are infinitely many responses to some operation. A type has *bounded* nondeterminism if there is some integer b so that there are at most b responses to any operation in any state. (Thus, the number of possible responses from each state is not only finite, but bounded.)

A wait-free algorithm requires each process to halt within a finite number of its own steps. In general, the actual number of steps depends on the inputs of the processes, the responses given to the processes by the objects, and the actual interleaving of the processes.

Consider a wait-free algorithm using objects with bounded nondeterminism. In this case, using König's Lemma, it is easy to show the following: For any choice of inputs to the processes, there is a nonnegative integer t so that every process takes no more than t steps in any execution where processes start with the specified inputs. Thus, we can express the maximum number of steps a process can take as a function of the inputs.

This, however, is not true if the algorithm uses objects with infinite nondeterminism. For example, consider a type that returns an arbitrary nonnegative integer to any invocation. Consider, further, the following algorithm that uses this object: Each process accesses the object, receives some response t, executes t "no-op" steps and halts. This algorithm is wait free in the sense that any process will halt after a finite number of its own steps. There is, however, no bound on the number of steps before halting.

Schenk defines an algorithm to be wait free if, for any input, there is a bound on the number of steps a process may take before it halts.[2] This choice of definition is crucial for his proof technique. Thus, two related questions are left open in [16]: (i) Is the Consensus hierarchy robust if we consider only types with *bounded* nondeterminism? (ii) Is the nondeterministic Consensus hierarchy robust if we use the more general definition of wait freedom? We answer the first question in the negative. This implies a negative answer for the second question as well: as noted above, the two definitions of wait freedom coincide in the context of bounded nondeterministic types.

1.3 Organisation of the paper

In Section 2 we define the model of computation used in this paper. In Section 3 we review the Consensus problem and introduce the Equality Negation problem. In Section 4 we derive some basic results relating the solvability of Equality Negation to the solvability of Consensus. In Section 5 we define the types negation and booster. In Section 6 we show how to solve Consensus for two processes using registers along with a negation and a booster object. In Sections 7 and 8 we prove that registers and negation objects alone, as well as registers and booster objects alone, cannot be used to solve Consensus for two processes. Section 9 concludes the paper.

[2]This definition of wait-freedom bears some similarity to, but differs from, Herlihy's notion of *bounded* wait-freedom (cf. [6]). In a bounded wait-free algorithm, there is a bound on the number of steps that is independent of the inputs. In Schenk's definition, the bound may depend on the inputs. In the most general definition, the number of steps may depend not only on the inputs but also on the responses given by the objects.

2 The model of computation

In this section we describe the model of computation we use in this paper. We caution that this model is designed to be as simple as possible while allowing us to express with reasonable precision the specific results of this paper. It is not rich enough to describe *all* results on wait-free synchronisation.

2.1 Types and objects

An *object type* 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 $\delta \subseteq Q \times OP \times RES \times Q$ is a *state transition relation*. The state transition relation describes the possible behaviours of objects of that type, 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'. If the state transition relation has the property that for any $q \in Q$ and $op \in OP$, there is at most one $res \in RES$ and $q' \in Q$ so that $(q, op, res, q') \in \delta$, then we say that the type is *deterministic*; otherwise, the type is *nondeterministic*.

For convenience we require each type to be *total*: For any $q \in Q$ and $op \in OP$ there is at least one $res \in RES$ and $q' \in Q$ such that $(q, op, res, q') \in \delta$. That is, the type can do *something* in response to any operation in any state. Further, we assume that, for any q, op and res, there is at most one q' so that $(q, op, res, q') \in \delta$. In other words, in response to an operation from a given state, the type does not lead to different states unless it returns different responses. All types we consider in this paper have these properties, so we can make these assumptions without loss of generality.

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 $P = (\Sigma, \Sigma_0, \Sigma_f, \nu, \tau)$, where Σ is a set of *states*; $\Sigma_0 \subseteq \Sigma$ is a set of *initial states*; $\Sigma_f \subseteq \Sigma$ is a set of *final states*; and the functions $\nu : (\Sigma \setminus \Sigma_f) \to \mathcal{O} \times OP$, and $\tau : (\Sigma \setminus \Sigma_f) \times RES \to \Sigma$ describe the interaction of the process with the objects. Intuitively, if P is in a nonfinal state $\sigma \in \Sigma$ and $\nu(\sigma) = \langle O, op \rangle$, 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. If $\mathcal{S}$ is a set of types, we say that $\mathbf{A}$ *uses* $\mathcal{S}$ if every object that $\mathbf{A}$ uses belongs to a type in $\mathcal{S}$. A *configuration* C of $\mathbf{A}$ is a function mapping every process in Π and object in $\mathcal{O}$ to one of that process' or object's states. 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 $P = (\Sigma, \Sigma_0, \Sigma_f, \nu, \tau)$ and C be a configuration, where the state of P in C is σ. If $\nu(\sigma) = \langle O, op \rangle$, we say that P has operation op on 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* $\mathbf{A}$ is a (finite or infinite) sequence of steps of $\mathbf{A}$'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, S' are schedules, $S \cdot S'$ denotes their concatenation.

We say that $\mathbf{A}$ is a *wait-free* algorithm if there is no infinite schedule S that is applicable to any initial configuration of $\mathbf{A}$. In other words, there is no execution of the algorithm in which any process can keep taking steps forever. This is so for any interleaving of steps, and even if some or even all other processes crash and stop taking steps. Throughout this paper we consider only wait-free algorithms, so we drop the qualification "wait-free", except when we want to draw special attention to this property.

2.3 Computation trees

Let $\mathbf{A}$ be an algorithm, and I be an initial configuration of $\mathbf{A}$. Consider a tree $\mathcal{T}$ whose nodes are finite schedules of $\mathbf{A}$ that are applicable to I (not necessarily all of them), so that the root of $\mathcal{T}$ is the empty schedule $S_\perp$, and there is an edge from node S to node S' only if $S' = S \cdot e$, for some step e. We say that $\mathcal{T}$ is a *computation tree of* $\mathbf{A}$ *from* I if for any node S of $\mathcal{T}$ and any process P, if P is not in a final state in $S(I)$, then S has a child $S \cdot e$ where e is a step of P. In the sequel we shall not distinguish between a node in a computation tree and the (finite) schedule corresponding to the node.

Informally, a computation tree represents executions of **A** starting from I for all possible interleavings of the processes. A node S represents a point in some execution; at that point the algorithm is in configuration $S(I)$. It is easy to see that if all object types used by **A** are deterministic, there is a unique computation tree from I. If, however, some objects are nondeterministic, there may be several computation trees.

A computation tree is *full* if its set of nodes is the entire set of finite schedules of **A** that are applicable to I. The full computation tree (of **A** from I) is unique, regardless of whether the objects used by **A** are deterministic. If **A** uses nondeterministic object types, we can think of the different computation trees as obtained from the full computation tree by pruning away branches that correspond to some (but not all) of the possible values returned to a process P by a nondeterministic object in an access made when the object was in a state that allows different responses to the operation applied by P.

We use the following notation and terminology about computation trees. Let $\mathcal{T}$ be a computation tree, S be a node in $\mathcal{T}$, and P be a process. The subtree of $\mathcal{T}$ that is rooted in S is denoted $\mathit{subtree}(S, \mathcal{T})$. The set of nodes $S \cdot e$ in $\mathcal{T}$, where e is a step of P, is denoted $\mathit{children}(P, S, \mathcal{T})$; these are the nodes reached from S by a single step of process P. We say that S_1 is a *P-sibling* of S_2 (in $\mathcal{T}$) if there is a node S of $\mathcal{T}$ so that $S_1, S_2 \in \mathit{children}(P, S, \mathcal{T})$.

3 Two Problems

In this section we review the Consensus problem and introduce Equality Negation, a problem that plays a central role in our results. In both problems, we consider the case of two processes.

3.1 The Consensus problem

In the Consensus problem, each process has a private initial value, drawn from the set $\{0, 1\}$, and must irrevocably decide either 0 or 1, so that the decisions of the processes are the same and equal to the initial value of some process. A Consensus algorithm must therefore satisfy the following three properties, starting from any one of its initial configurations.

Termination: If a process reaches a final state, then it must have decided 0 or 1.[3]

Agreement: If both processes decide, then they decide the same value.

Validity: The decision of any process is the initial value of some process.

[3] Termination is usually expressed as a liveness property: if a process is correct then it eventually decides. In our case, we can get away with this alternate statement (which defines a safety property) thanks to the implicit assumption that algorithms are wait-free.

3.2 The Equality Negation problem

In the Equality Negation problem, each process has a private initial value, drawn from the set $\{0, 1, 2\}$ (instead of $\{0, 1\}$ in Consensus), and must irrevocably decide either 0 or 1 so that the decisions of the processes are the same if and only if the initial values of the processes are different. An Equality Negation algorithm must therefore satisfy the following two properties, starting from any one of its initial configurations.

Termination: If a process reaches a final state, then it must have decided 0 or 1.

Negation: If both processes decide, then their decisions are the same *if and only if* their initial values are different.

The following lemma expresses a simple property of computation trees of Consensus and Equality Negation algorithms.

Lemma 1 *Let* $\mathbf{A}$ *be any Consensus or Equality Negation algorithm for processes* P_0 *and* P_1*,* I *be any initial configuration of* $\mathbf{A}$*,* $\mathcal{T}$ *be any computation tree of* $\mathbf{A}$ *from* I*, and* S *be any node in* $\mathcal{T}$*. For each* $k \in \{0, 1\}$*, there exists a solo schedule* S_k *of process* P_k *such that* $S \cdot S_k$ *in a node in* $\mathcal{T}$ *and* P_k *has decided in* $S \cdot S_k(I)$*.*

PROOF. Immediate from the definition of computation tree, the fact that algorithms are wait-free, and the Termination requirement of Consensus or Equality Negation. □

4 On the solvability of Equality Negation

In this section we present some results concerning the solvability of Equality Negation. On the one hand, we prove that solving Equality Negation is no harder than solving Consensus (cf. Section 4.1). On the other hand, we also prove that solving Equality Negation is not too easy: it can't be done using just registers (cf. Section 4.2). These results will be used in Section 8.

4.1 Consensus vs. Equality Negation

We show that solving Equality Negation for two processes is no harder than solving Consensus for two processes: Given an *arbitrary* algorithm $\mathbf{A}_c$ that solves Consensus for processes P_0 and P_1, we show how to construct (using $\mathbf{A}_c$ as a subroutine) an algorithm that solves Equality Negation for processes Q_0 and Q_1. The Equality Negation algorithm is shown in pseudocode in Figure 1. Following are some conventions used in the pseudocode of this and other algorithms. We use capital letters for names of shared objects (including registers), and lower-case letters for names of local variables. We often use the same name for local variables of different processes; such variables are assumed to be distinct. For example, local variable v of Q_0 and local variable v of Q_1 in Figure 1

```
Shared: objects used in A_c, each has the initial state specified by A_c
        R_0, R_1: register, each initialised to ⊥

For process Q_k, k ∈ {0,1}
1  R_k := initial value of Q_k
2  v := Execute(P_k, A_c, k)
3  if v = k then decide k
4  else
5      if R_k̄ = R_k then decide k
6      else decide k̄
```

Figure 1: Solving Equality Negation for 2 processes using a Consensus algorithm $\mathbf{A}_c$.

are distinct. This convention emphasises the symmetric nature of the algorithms while keeping the notation relatively simple. To access a register R, we place R on the left-hand side of an assignment statement to indicate a WRITE operation applied to R, and use the name R in any other context to indicate a READ operation applied to R. If $\mathbf{A}$ is a Consensus or Equality Negation algorithm and P is a process of $\mathbf{A}$, the procedure Execute$(P, \mathbf{A}, \ell)$ causes the caller to simulate process P in algorithm $\mathbf{A}$ using k as its initial value, and returns the decision of P in the simulated execution of that algorithm.

Lemma 2 *If $\mathbf{A}_c$ is a Consensus algorithm for processes P_0 and P_1, then the algorithm in Figure 1 solves Equality Negation for processes Q_0 and Q_1.*

PROOF. It is obvious that the algorithm in Figure 1 is wait-free (since $\mathbf{A}_c$ is), and that it satisfies Termination. To show that it satisfies the Negation property, consider any execution in which both Q_0 and Q_1 decide. Let v_k be the value that process Q_k assigned to its local variable v in line 2. By the Agreement property of $\mathbf{A}_c$, $v_0 = v_1 = d$, for some $d \in \{0, 1\}$. Process P_d decides d (cf. line 3). Process $P_{\overline{d}}$[4] decides as follows: If the initial values of Q_d and $Q_{\overline{d}}$ are the same (hence $R_d = R_{\overline{d}}$ in line 5), then $P_{\overline{d}}$ decides $\overline{d}$; otherwise, $P_{\overline{d}}$ decides d (cf. line 6). In other words, P_d and $P_{\overline{d}}$ decide the same value if and only if their initial values are distinct. □

Corollary 1 *Let $\mathcal{S}$ be a set of object types that includes* register. *If Consensus for two processes is solvable using $\mathcal{S}$, then Equality Negation for two processes is also solvable using $\mathcal{S}$.*

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

Shared: all registers used in $\mathbf{A}_{en}$, each has the initial value specified by $\mathbf{A}_{en}$
R_0, R_1: register, each initialised to $\perp$

For process Q_0
1 $R_0 :=$ initial value of Q_0
2 **if** $R_1 = \perp$ **then**
3 $\quad w :=$ Execute$(P_0, \mathbf{A}_{en}, 0)$
4 $\quad$ **if** $w = v$ **then decide** R_0
5 $\quad$ **else decide** R_1
6 **else**
7 $\quad w :=$ Execute$(P_0, \mathbf{A}_{en}, 1)$
8 $\quad$ **if** $w = v$ **then decide** R_1
9 $\quad$ **else decide** R_0

For process Q_1
1 $R_1 :=$ initial value of Q_1
2 **if** $R_0 = \perp$ **then**
3 $\quad w :=$ Execute$(P_1, \mathbf{A}_{en}, 0)$
4 $\quad$ **if** $w = v$ **then decide** R_1
5 $\quad$ **else decide** R_0
6 **else**
7 $\quad w :=$ Execute$(P_1, \mathbf{A}_{en}, 1)$
8 $\quad$ **if** $w = v$ **then decide** R_0
9 $\quad$ **else decide** R_1

Figure 2: Solving Consensus for processes Q_0 and Q_1 using $\mathbf{A}_{en}$, when $v_0 = v_1 = v$.

4.2 Unsolvability of Equality Negation using only registers

We now show that Equality Negation for two processes is not solvable using registers alone. The proof is by contradiction. Suppose that there is an Equality Negation algorithm, $\mathbf{A}_{en}$, for process P_0 and P_1 that uses only registers. We show that, using $\mathbf{A}_{en}$ as a subroutine and two additional registers, we can construct a Consensus algorithm for two processes Q_0 and Q_1. Thus, we have a Consensus algorithm for two processes that uses only registers, contradicting the well-known fact that no such algorithm exists [13].

For each $k \in \{0,1\}$, consider a solo execution of process P_k in algorithm $\mathbf{A}_{en}$, where the initial value of P_k is 0. Since $\mathbf{A}_{en}$ uses only registers (which are deterministic), the solo execution of P_k with initial value 0 is uniquely defined. Let v_k be the decision of P_k in this solo execution. The Consensus algorithm that uses $\mathbf{A}_{en}$ depends on whether $v_0 = v_1$. If $v_0 = v_1$, the Consensus algorithm is shown in Figure 2. If $v_0 \neq v_1$, the Consensus algorithm is shown in Figure 3.

Lemma 3 *If $v_0 = v_1 = v$, for some $v \in \{0,1\}$, then the algorithm in Figure 2 solves Consensus for two processes.*

Proof. It is obvious that the algorithm in Figure 2 is wait-free (since $\mathbf{A}_{en}$ is), and that it satisfies Termination.

To prove that the algorithm satisfies Validity, it suffices to show that if process Q_k decides by reading register R_m, then Q_m has previously written its initial value into R_m. This is immediate if Q_k decides by lines 4, 8 or 9. It remains to cover the case where Q_k decides by line 5. In this case, we want to prove that before Q_k decides $R_{\overline{k}}$ in line 5, $Q_{\overline{k}}$ has written its initial value into $R_{\overline{k}}$ in line 1. For this, it is enough to show that the execution of P_k that Q_k simulated in line 3 is not a solo P_k execution of $\mathbf{A}_{en}$. (Because, in that case, Q_k executed line 3 concurrently with $Q_{\overline{k}}$'s execution of line 3 or line 7. Hence, $Q_{\overline{k}}$ executed line 1 before Q_k executed line 5.) To see why the execution of P_k that Q_k simulated is not a solo P_k execution, suppose the contrary. Then, by definition of v_k, P_k would have decided $v_k = v$ in that execution, and the value that Q_k assigned to its local variable w in line 3 would have been equal to v. But then Q_k would be deciding by line 4, not 5.

To prove that the algorithm satisfies Agreement, consider an execution in which both Q_0 and Q_1 decide. Let w_0 and w_1 be the values of local variable w of Q_0 and Q_1, respectively, assigned in line 3 or 7. Now consider the execution of algorithm $\mathbf{A}_{en}$ by P_0 and P_1 that Q_0 and Q_1 simulated in line 3 or 7. Note that, in this execution, P_0 and P_1 cannot both have initial value 0: for this to happen, each of Q_0 and Q_1 must execute its line 2 before either executes its line 1, which is impossible. Given this, there are three remaining possibilities for the initial values of P_0 and P_1 in the simulated execution of $\mathbf{A}_{en}$. We shall prove that in each of these cases, both Q_0 and Q_1 decide by reading R_0, or both decide by reading R_1.

Case 1. *P_0 has initial value 0 and P_1 has initial value 1:* By the Negation property of $\mathbf{A}_{en}$, P_0 and P_1 will decide the same value and hence $w_0 = w_1 = w^*$, for some $w^* \in \{0, 1\}$. If $w^* = v$, Q_0 and Q_1 both decide R_0 (cf. line 4 of Q_0 and line 8 of Q_1); otherwise, they both decide R_1 (cf. line 5 of Q_0 and line 9 of Q_1).

Case 2. *P_0 has initial value 1 and P_1 has initial value 0:* Similar to Case 1.

Case 3. *P_0 and P_1 both have initial value 1:* By the Negation property of $\mathbf{A}_{en}$, P_0 and P_1 will decide different values. Thus, either $w_0 = v$ and $w_1 \neq v$, or $w_0 \neq v$ and $w_1 = v$. In the former case, Q_0 and Q_1 decide R_1 (cf. line 8 of Q_0 and line 9 of Q_1). In the latter case, Q_0 and Q_1 decide R_0 (cf. line 9 of Q_0 and line 8 of Q_1).

Thus, both processes decide by reading the same register. By Validity, a process decides by reading R_k only after Q_k has written its initial value into R_k. Since only one value is written into R_k, Agreement follows. □

Lemma 4 *If $v_0 \neq v_1$, then the algorithm in Figure 3 solves Consensus for two processes.*

Proof. It is obvious that the algorithm in Figure 3 is wait-free (since $\mathbf{A}_{en}$ is), and that it satisfies Termination. The proof that the algorithm satisfies Validity is similar to the corresponding part of the proof of Lemma 3 and is omitted.

To prove that the algorithm satisfies Agreement, consider an execution in which both Q_0 and Q_1 decide. Let w_0 and w_1 be the values of local variable w of Q_0 and Q_1, respectively, assigned in line 3 or 5. Now consider the execution of algorithm $\mathbf{A}_{en}$ by P_0 and P_1 that Q_0 and Q_1 simulated in line 3 or 5. As in the corresponding part of the proof of Lemma 3, in this execution, P_0 and P_1 cannot both have initial value 0. In all three of the remaining possibilities for the initial values of P_0 and P_1, those values are distinct.

Shared: R_0, R_1: register, each initialised to $\perp$

For process Q_0

1 $R_0 :=$ initial value of Q_0
2 **if** $R_1 = \perp$ **then**
3 $\quad w := \text{Execute}(P_0, \mathbf{A}_{en}, 0)$
4 **else**
5 $\quad w := \text{Execute}(P_0, \mathbf{A}_{en}, 1)$
6 **if** $w = v_0$ **then decide** R_0
7 **else decide** R_1

For process Q_1

1 $R_1 :=$ initial value of Q_1
2 **if** $R_0 = \perp$ **then**
3 $\quad w := \text{Execute}(P_1, \mathbf{A}_{en}, 0)$
4 **else**
5 $\quad w := \text{Execute}(P_1, \mathbf{A}_{en}, 2)$
6 **if** $w = v_1$ **then decide** R_1
7 **else decide** R_0

Figure 3: Solving Consensus for processes Q_0 and Q_1 using $\mathbf{A}_{en}$, when $v_0 \neq v_1$.

By the Negation property of Equality Negation, P_0 and P_1 decide the same value in the simulated execution of $\mathbf{A}_{en}$. Hence, $w_0 = w_1 = w^*$, for some $w^* \in \{0, 1\}$. If $w^* = v_0$ (and therefore $w^* \neq v_1$), then both Q_0 and Q_1 decide R_0 (cf. line 6 of Q_0 and line 7 of Q_1). If $w^* \neq v_0$ (and therefore $w^* = v_1$), then both Q_0 and Q_1 decide R_1 (cf. line 7 of Q_0 and line 6 of Q_1).

Thus, both processes decide by reading the same register. By Validity, a process decides by reading R_k only after Q_k has written its initial value into R_k. Since only one value is written into R_k, Agreement follows. □

Theorem 1 *There is no algorithm that solves Equality Negation for two processes using registers alone.*

PROOF. If an Equality Negation algorithm for two processes that uses only registers existed, by Lemmata 3 and 4 there would also exist a Consensus algorithm for two processes that uses only registers. As shown in [13], however, no such Consensus algorithm exists. □

5 Types negation and booster

We define two nondeterministic object types, named negation and booster, that will be used later to establish the result that the Consensus hierarchy is not robust. Both of these types have *bounded* nondeterminism; indeed, each has only finitely many states.

The specification of negation is given in Figure 4. The pseudocode describes the state transition relation of negation. In addition to standard programming constructs, we use the function "Choose" which takes as a parameter a set V of integers and returns an arbitrary value in V. We also assume that the present state of the type is recorded in variable *current_state*. To understand the specification, it is useful to keep in mind

Set of states: $\{☺, ☹\} \cup \{\langle v, u\rangle : v \in \{0,1,2\}, u \in \{0,1\}\}$.
Set of operations: $\{\text{NEGATE}(i) : i \in \{0,1,2\}\}$.
Set of responses: $\{0,1\}$.

NEGATE(i), $i \in \{0,1,2\}$, return either 0 or 1
  case *current_state*
    ☹: **return** Choose($\{0,1\}$)
    ☺: $w \leftarrow$ Choose($\{0,1\}$); *current_state* $\leftarrow \langle i, w\rangle$; **return** w
    $\langle v, u\rangle$:
      current_state $\leftarrow$ ☹
      if $v = i$ **then return** $\overline{u}$
      else return u

Figure 4: Specification of type negation.

the following intuitive explanation of the type. There is only one kind of operation, named NEGATE, which takes a parameter in $\{0,1,2\}$, and returns a value in $\{0,1\}$. The type is such that two NEGATE operations with different parameters will return the same value, while two NEGATE operations with the same parameter will return different values. Once two NEGATE operations have been applied, the object becomes "upset" and returns arbitrarily either 0 or 1 to any subsequent NEGATE operations. Thus, negation is a type designed to solve Equality Negation between two processes in a very direct way: A process with initial value $i \in \{0,1,2\}$, simply applies the operation NEGATE(i) and decides the value returned.

To achieve this behaviour, type negation has eight states: the *fresh* state, denoted ☺; the *upset* state, denoted ☹; and six states of the form $\langle v, u\rangle$, where $v \in \{0,1,2\}$ and $u \in \{0,1\}$. The intended meaning of state $\langle v, u\rangle$, assuming that the object was initially in the fresh state, is that v was the parameter of the first NEGATE operation applied and u was the value returned to that operation. With this information, the object knows all it needs to return the appropriate value to the second NEGATE operation.

The specification of booster is shown in Figure 5. To understand the specification, it is useful to keep in mind the following intuitive explanation of the type. There are two kinds of operations that can be applied to a booster object: ENROLL and REVEAL. Both operations take as a parameter a value in $\{0,1\}$. Typically, the first thing process P_k must do to a booster object is to apply ENROLL(k) to "register" itself with the object. Intuitively, the parameter of an ENROLL operation is a process id. The object maintains a secret: the identity of the first process to have enrolled. The object will reveal this secret to a process that subsequently accesses it with the REVEAL operation, provided that the process can "prove" to the object that it can solve Equality Negation. Here is how this works.

Set of states: $\{☺, ☹\} \cup \{\, \langle v_0, v_1, u, d\rangle : v_0, v_1 \in \{\perp, 0, 1, 2\} \wedge u \in \{\perp, 0, 1\} \wedge d \in \{0, 1\}\,\}$.
Set of operations: $\{\, \text{ENROLL}(i), \text{REVEAL}(i) : i \in \{0, 1\}\,\}$.
Set of responses: $\{0, 1, 2\}$.

ENROLL(i), $i \in \{0, 1\}$, return a value in $\{0, 1, 2\}$
    $v \leftarrow$ Choose($\{0, 1, 2\}$)
    case *current_state*
        ☺: **if** $i = 0$ **then** *current_state* $\leftarrow \langle v, \perp, \perp, 0\rangle$
            else *current_state* $\leftarrow \langle \perp, v, \perp, 1\rangle$
        $\langle v_0, v_1, u, d\rangle$:
            if $v_i \neq \perp$ **or** $v_{\bar{i}} = \perp$ **then** *current_state* $\leftarrow$ ☹
            else if $i = 0$ **then** *current_state* $\leftarrow \langle v, v_1, u, d\rangle$
            else *current_state* $\leftarrow \langle v_0, v, u, d\rangle$
    return v

REVEAL(i), $i \in \{0, 1\}$, return a value in $\{0, 1\}$
    case *current_state*
        ☺: *current_state* $\leftarrow$ ☹; **return** Choose($\{0, 1\}$)
        ☹: **return** Choose($\{0, 1\}$)
        $\langle v_0, v_1, u, d\rangle$:
            if $u = \perp$ **then**
                current_state $\leftarrow \langle v_0, v_1, i, d\rangle$; **return** d
            else
                current_state $\leftarrow$ ☹
                if $v_0 = \perp$ **or** $v_1 = \perp$ **then return** Choose($\{0, 1\}$)
                else
                    if ($v_0 \neq v_1$ **and** $i = u$) **or** ($v_0 = v_1$ **and** $i \neq u$) **then return** d
                    else return Choose($\{0, 1\}$)

Figure 5: Specification of type booster.

In response to the ENROLL(k) operation it applies, P_k receives a value in $\{0, 1, 2\}$. This value is a "challenge" given by the object, which the process must use as its input value to solve Equality Negation. After solving Equality Negation and deciding a value $d \in \{0, 1\}$, P_k applies the operation REVEAL(d) to the booster object. If the parameter of a REVEAL operation is a legitimate decision for the challenge input(s) returned by the ENROLL operation(s) applied to the object so far, the object will respond to the REVEAL by returning the secret: the identity of the first process to have enrolled. In the opposite case, the REVEAL operation returns an arbitrary value in $\{0, 1\}$.

There is an etiquette that a process must follow in accessing a booster object: First enroll and obtain a challenge input value, then get the secret by presenting to the object a decision for Equality Negation that is legitimate given the challenge inputs provided by the object. Any deviation from this etiquette will result in the object becoming "upset". Once upset, the object responds to any operation applied to it by returning an arbitrary value.

To achieve this behaviour, the negation type has the following states: A *fresh* state, denoted ☺; an *upset* state, denoted ☹; the remaining states are of the form $\langle v_0, v_1, u, d\rangle$, where $v_0, v_1 \in \{\perp, 0, 1, 2\}$, $u \in \{\perp, 0, 1\}$, and $d \in \{0, 1\}$. The intended meaning of these is as follows (assuming that the object is initialised to the fresh state):

- v_k, for $k \in \{0, 1\}$, if not equal to $\perp$, is the challenge input (for Equality Negation) given to process P_k (more accurately, the value returned to ENROLL(k) was v_k).
- d is the "secret" — the identity of the first process that enrolled (more accurately, the parameter of the first ENROLL operation).
- u is the decision reached for Equality Negation by the first process that tries to get the secret (more accurately, the parameter of the first REVEAL operation).

With this information, the object knows all it needs to respond to operations appropriately.

6 The main result

In this section we prove that the Consensus hierarchy is not robust, taking for granted the fact that negation and booster have Consensus number one (this will be shown in Sections 7 and 8). To do so, we give a Consensus algorithm for two processes that uses only negation and booster objects, and registers. The algorithm is shown in Figure 6. The notation "Apply(P, op, O)" denotes the procedure by which process P applies operation op to object O; the procedure returns the response of the object to that operation.

Before presenting its correctness proof, we briefly describe how the algorithm works. From the intuitive explanations of the type negation and booster, we can see that a negation object allows two processes to solve Equality Negation, and a booster object allows two processes that can solve Equality Negation to agree on which of the two was the first to enroll to the booster object. Using a negation and a booster object, as well as registers, two processes can thus solve Consensus as follows (see Figure 6): Each process P_k writes its initial value (for Consensus) into a register R_k. P_k then enrolls to the booster object (by applying ENROLL(k)) and obtains a challenge input, say v, with which to solve Equality Negation. P_k does so by applying NEGATE(v) to the negation object. Let w be the value returned by NEGATE(v) (i.e., the decision of P_k in Equality Negation with v as its input). By applying REVEAL(w), P_k determines the id x of the first process to have applied the ENROLL operation to the booster object, and decides the initial value of that process, written in R_x. Since both processes agree on who applied the first ENROLL operation, they both decide the initial value of the same process.

Shared: O_b: booster, initialised to ☺
O_n: negation, initialised to ☺
R_0, R_1: register, initialised to $\perp$

For processes P_k, $k \in \{0,1\}$
1 $R_k :=$ initial value of P_k
2 $v := \text{Apply}(P_k, \text{ENROLL}(k), O_b)$
3 $w := \text{Apply}(P_k, \text{NEGATE}(v), O_n)$
4 $x := \text{Apply}(P_k, \text{REVEAL}(w), O_b)$
5 **decide** R_x

Figure 6: A Consensus algorithm for two processes using $\{\mathsf{negation}, \mathsf{booster}, \mathsf{register}\}$.

Theorem 2 *The algorithm in Figure 6 solves the Consensus problem for two processes using only* negation *objects,* booster *objects, and registers.*

Proof. It is obvious that the algorithm uses only objects of the allowed types, that it is wait-free and that it satisfies Termination. Next we prove that the algorithm satisfies Validity. It suffices to show that if process P_k decides by reading register R_d, then P_d has previously written its initial value into R_d. This is clearly true if $d = k$ or $P_{\overline{k}}$ has taken at least one step before P_k decides, since the first step of each process in the algorithm is to write its initial value into its register. It remains to consider the case when P_k decides before $P_{\overline{k}}$ executes any steps. We shall show that, in this case, P_k decides by reading R_k. By the specification of type booster, it is easy to verify that just before P_k applies the REVEAL operation to $\mathcal{O}_b$ in line 4, O_b is in a state of the form $\langle *, *, \perp, k\rangle$. Since a REVEAL operation to a booster object in a state of the form $\langle *, *, \perp, k\rangle$ returns k, the value assigned to local variable x of P_k in line 4 is k. Thus, by line 5, P_k decides by reading R_k.

Finally, to prove that the algorithm satisfies Agreement, consider an execution in which both processes decide. For each $k \in \{0,1\}$, let v_k, w_k and x_k be the values assigned to local variables v, w and x, respectively, of process P_k. After both processes have applied the ENROLL operation to O_b in line 2, O_b is in state $\langle v_0, v_1, u, d\rangle$, where d is such that P_d applied the ENROLL operation to O_b in line 2 before $P_{\overline{d}}$. By the specification of negation, $w_0 = w_1$ if and only if $v_0 \neq v_1$. It then follows from the specification of booster that $x_0 = x_1 = d$. By line 5, both processes decide the value they read in R_d. By Validity, R_d is read only after being written by P_d. Since R_d is written only once, both processes decide the same value. □

In Sections 7 and 8 we prove that each of negation and booster has Consensus number one (Theorems 3 and 4). Theorem 2 and the universality of Consensus (cf. [7]) imply that, in a system of two processes, we can implement any type using booster and negation.

This means that by combining the two types, `negation` and `booster`, that are in level one of the Consensus hierarchy, we can implement any type at level two. Thus, we get the main result of this paper:

Corollary 2 *The Consensus hierarchy is not robust.*

7 Type negation has Consensus number one

In this section we prove that type `negation` has Consensus number one — i.e., that Consensus for two processes cannot be solved using only `negation` objects and registers. The proof makes extensive use of computation trees, defined in Section 2.3.

7.1 Valence of nodes in Consensus computation trees

Let $\mathbf{A}_c$ be a Consensus algorithm, I be an initial configuration of $\mathbf{A}_c$, and $\mathcal{T}$ be a computation tree of $\mathbf{A}_c$ from I. A node S of $\mathcal{T}$ is *bivalent* if it has descendents S_0 and S_1 so that some process decides 0 in $S_0(I)$ and some process decides 1 in $S_1(I)$. S is *v-valent*, for $v \in \{0,1\}$, if it has no descendent S' so that some process decides $\overline{v}$ in $S'(I)$. S is *univalent* if it is either 0-valent or 1-valent. By the Termination property of Consensus, every node of $\mathcal{T}$ is is either univalent or bivalent, and every leaf is univalent. Informally, a univalent node represents a partial execution of $\mathbf{A}_c$ in which the outcome of Consensus has been "sealed" — even if no process has actually decided yet.

The *valence* of a node S refers to whether S is bivalent or univalent and, in the latter case, whether it is 0-valent or 1-valent. We say that two univalent nodes *have opposite valence* if one is 0-valent and the other is 1-valent. As noted in Section 2.3, if $\mathbf{A}_c$ uses nondeterministic objects, there may be many computation trees of $\mathbf{A}_c$ from I. A node that appears in two different computation trees may have a different valence in one than in the other, because the valence of a node S in a computation tree $\mathcal{T}$ depends on the subset of S's descendents in the *full* computation tree that are actually present in $\mathcal{T}$.

7.2 The bivalence argument and some limitations

There is a standard technique for proving that there is no Consensus algorithm for n processes using a certain set of object types, known as "the bivalence argument". This technique goes back to the ground-breaking result of Fischer *et al.* [5], and was further adapted by [4, 13, 7]. In this subsection we sketch the main ingredients of this technique for two processes, and call attention to a difficulty that arises when it is applied to nondeterministic object types.

Assume, for contradiction, that there is a Consensus algorithm $\mathbf{A}_c$ for P_0 and P_1 that uses only objects of the given types. Consider the full computation tree $\mathcal{F}$ of this algorithm from the initial configuration I in which the two processes have different initial values. By the choice of the initial configuration, it is easy to see that the root of $\mathcal{F}$ is bivalent. Since the root is bivalent and every leaf is univalent, $\mathcal{F}$ contains a *critical* node S — i.e., a node that is bivalent but all of whose children are univalent. Furthermore, some of the (univalent) children of the critical node S must be 0-valent and some must be 1-valent (because S itself is bivalent) in $\mathcal{F}$.

If the object types used by $\mathbf{A}_c$ are deterministic, then S has exactly two children, say $S \cdot e_0$ and $S \cdot e_1$, where e_k is a step of process P_k. Thus, in this case, we can say that *the outcome of Consensus after S is determined by which one of the two processes takes the next step.* It is then argued that e_0 and e_1 both access the same object O. (Otherwise, if each of the two processes applies its next step, both processes and all objects would be in exactly the same state, regardless of which of the two processes took the step first; this contradicts the fact that the identity of the process that takes the first step after S determines the outcome of Consensus.) Finally, by a case analysis involving the types of operations applied in steps e_0 and e_1, the following is shown: One of the two processes, P_k, has a solo schedule S_k so that (i) $S \cdot S_k$ and $S \cdot e_{\overline{k}} \cdot S_k$ are both nodes in $\mathcal{F}$, and (ii) P_k has decided in $S \cdot S_k(I)$. Since P_k is in the same state in $S \cdot S_k(I)$ as in $S \cdot e_{\overline{k}} \cdot S_k(I)$, it must have decided the same value in both configurations. This contradicts the fact that the outcome of Consensus is determined by which of the two processes takes the next step after S.

This type of argument runs into difficulties when applied to algorithms that use *nondeterministic* types. Specifically, we can no longer assert that at the critical node S, the outcome of Consensus is determined by which of the two processes takes the next step. Since the object types are nondeterministic, it is possible that *each* of the two processes can lead to both a 0-valent and a 1-valent node, depending on the response of the object accessed.

To appreciate this point in more concrete terms, suppose that the Consensus algorithm $\mathbf{A}_c$ uses negation objects and that the critical node S is such that in configuration $S(I)$ each process P_k has a NEGATE(k) operation pending to a negation object O in the fresh state. Suppose, further, that P_k decides the value returned by the NEGATE(k) operation (see Figure 7). Now the step of process P_k that causes it to decide v, namely e_k^v, is not applicable after the other process, $P_{\overline{k}}$, has applied the step that causes it to decide the opposite value, $\overline{v}$, for all $k \in \{0,1\}$ and all $v \in \{0,1\}$. This means that the we cannot derive a contradiction in the style of the bivalence argument.

Despite this, it turns out that the negation type is not powerful enough to solve Consensus between two processes. Unfortunately, however, the standard bivalence argument cannot be used to prove this fact. Instead, we use a modified version of the bivalence argument, based on a restricted computation tree. This is described in the following subsection.

7.3 The pruned computation tree

In the remainder of this section assume, for contradiction, that $\mathbf{A}_c$ is a Consensus algorithm for processes P_0 and P_1 that uses only negation objects and registers. Let I be the

For $k, v \in \{0,1\}$, $e_k^v = (P_k, \text{NEGATE}(k), O, v)$.

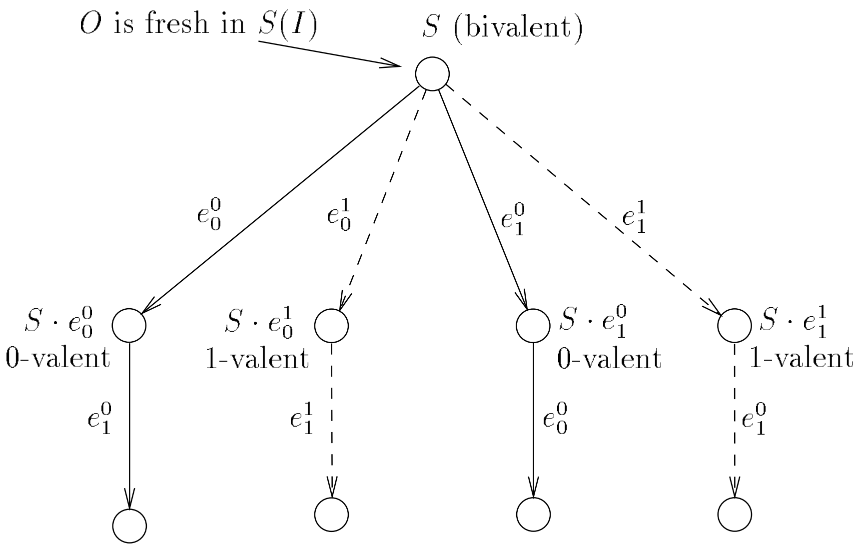

Figure 7: A critical node in $\mathcal{F}$.

initial configuration of $\mathbf{A}_c$ where P_0 and P_1 have initial values 0 and 1, respectively, and $\mathcal{T}$ be any computation tree of $\mathbf{A}_c$ from I. Consider a nonroot node S in $\mathcal{T}$, and let P be the process that takes the last step in S. We say that S is an *avoidable* 1-*valent node* in $\mathcal{T}$ if:

- S is 1-valent and its parent is bivalent in $\mathcal{T}$;
- S has a P-sibling S' that is not 1-valent in $\mathcal{T}$; and
- for any node S'' in the subtree rooted at S', if S'' is 1-valent and its parent is bivalent in $\mathcal{T}$, then S'' has a Q-sibling that is not 1-valent in $\mathcal{T}$, where Q is the process that takes the last step in S''.

The intuition behind this definition is the following (see Figure 8). An avoidable 1-valent node S is reached from its bivalent parent $\hat{S}$ by a step of some process P; that step seals the outcome of Consensus to be 1, in the sense that it takes us from a bivalent node to a 1-valent one. Furthermore, S has a P-sibling S' that is not 1-valent; this means that the object accessed by the operation of P that is pending in $\hat{S}(I)$, *could* have returned a value that would leave open the possibility of the outcome of Consensus being 0. Finally, *any* subsequent "decisive" step (to a descendent S'' of S') that would seal the outcome of Consensus to be 1, is taken by a process Q in an access to an object which,

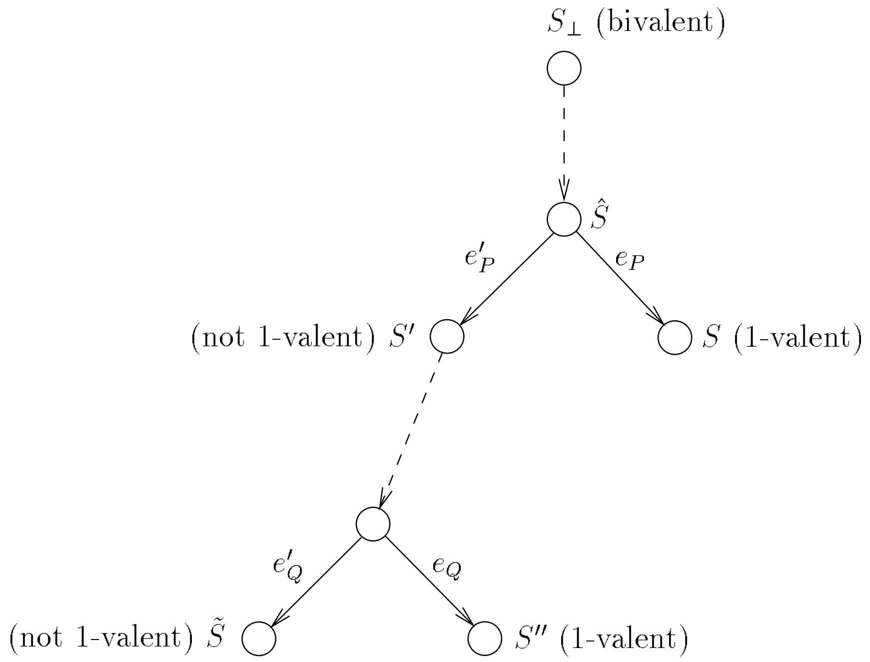

Figure 8: An avoidable 1-valent node in $\mathcal{T}$.

again, *could* have returned a value that would leave open the possibility of the outcome of Consensus being 0. In other words, starting with a step of P from $\hat{S}$ (the bivalent parent of S), outcome 1 could *always* be avoided by fortuitous choice of the values returned by the objects accessed by the processes, *regardless of the order in which the processes are scheduled to take steps.* As a concrete illustration, the computation tree shown in Figure 7 contains two avoidable 1-valent nodes, namely $S \cdot e_0^1$ and $S \cdot e_1^1$.

The *pruned computation tree of* $\mathbf{A}_c$ *from* I, denoted $\mathcal{L}$, is the tree obtained from the *full* computation tree $\mathcal{F}$ of $\mathbf{A}_c$ from I by pruning all its avoidable 1-valent nodes — i.e., removing them and their descendents. $\mathcal{L}$ is, in fact, a computation tree: if a node S is pruned, then the parent of S, $\hat{S}$, has another child, S', that is not pruned, so that the same process P that takes the step leading from $\hat{S}$ to S, also takes the step leading from $\hat{S}$ to S'. In other words, the pruning of S will not inhibit P from taking the next step after $\hat{S}$.

Intuitively, pruning the avoidable 1-valent nodes from the full computation tree amounts to restricting the nondeterministic behaviour of objects: If the objects could "conspire" to lead the computation to outcome 0 regardless of how processes are scheduled to take their steps, we restrict their behaviour to ensure that outcome 0 is actually reached.

The following lemma states some basic properties of the pruned computation tree $\mathcal{L}$.

Lemma 5 *Let S be any node in the full computation tree $\mathcal{F}$. Then*

(a) S is not a node in $\mathcal{L}$ if and only if S is a descendent of an avoidable 1-valent node in $\mathcal{F}$. (By convention, a node is a descendent of itself.)

(b) If S is 1-valent in $\mathcal{F}$ and is also in $\mathcal{L}$, then S is 1-valent in $\mathcal{L}$.

(c) If S is 0-valent in $\mathcal{F}$, then it is in $\mathcal{L}$ and is 0-valent in $\mathcal{L}$.

(d) If S is in $\mathcal{L}$ and $children(P, S, \mathcal{F}) = \{S'\}$ (N.B.: a singleton), then S' is in $\mathcal{L}$.

(e) If S is in $\mathcal{L}$ and is 1-valent in $\mathcal{L}$, then $subtree(S, \mathcal{L}) = subtree(S, \mathcal{F})$.

PROOF. (a)–(d) are immediate consequences of the definition of $\mathcal{L}$. For (e), suppose that S is 1-valent in $\mathcal{L}$. S cannot have any 0-valent descendents in $\mathcal{F}$ (otherwise, by (c), any such descendents would exist in $\mathcal{L}$ and be 0-valent in $\mathcal{L}$, contradicting that S is 1-valent in $\mathcal{L}$). Thus, S is 1-valent in $\mathcal{F}$. Consequently, all nodes in $subtree(S, \mathcal{F})$ are 1-valent, and therefore none of them can be an avoidable 1-valent node. Hence, by (a), all nodes in $subtree(S, \mathcal{F})$ are in $\mathcal{L}$, proving that $subtree(S, \mathcal{L}) = subtree(S, \mathcal{F})$. $\square$

7.4 The impossibility result

The proof that no algorithm can solve Consensus between two processes using only negation objects and registers is a bivalence argument based on the pruned computation tree $\mathcal{L}$. The only remaining tricky point is to ensure that the computation tree nodes that exhibit the contradictory behaviour required by the bivalence argument have not been pruned away! We proceed with the proof by establishing some additional properties of $\mathcal{L}$.

Lemma 6 *Let S be any node in $\mathcal{L}$ and P be any process such that $|children(P, S, \mathcal{F})| = 2$, each node in $children(P, S, \mathcal{L})$ is univalent and one of them is 1-valent in $\mathcal{L}$. Then $children(P, S, \mathcal{L}) = children(P, S, \mathcal{F})$ and both nodes in $children(P, S, \mathcal{L})$ are 1-valent in $\mathcal{L}$.*

PROOF. Let $children(P, S, \mathcal{F}) = \{S', S''\}$. By the hypothesis of the lemma we may assume, without loss of generality, that $S' \in children(P, S, \mathcal{L})$ and S' is 1-valent in $\mathcal{L}$. It remains to show that S'' is also in $\mathcal{L}$ and is 1-valent in $\mathcal{L}$.

S'' is in $\mathcal{L}$: Since S' is 1-valent in $\mathcal{L}$, By Lemma 5(e), $subtree(S', \mathcal{L}) = subtree(S', \mathcal{F})$ and thus S' is 1-valent in $\mathcal{F}$. Thus, S'' is not an avoidable 1-valent node in $\mathcal{F}$. Furthermore, it cannot be a proper descendent of one (because then so would S', and by Lemma 5(a) S' would not be in $\mathcal{L}$). Therefore, by Lemma 5(a), S'' is in $\mathcal{L}$.

S'' is 1-valent in $\mathcal{L}$: Since S' is 1-valent in $\mathcal{F}$ and is also in $\mathcal{L}$, there is a node in $subtree(S'', \mathcal{F})$ that is 1-valent in $\mathcal{F}$ but is not a descendent of an avoidable 1-valent node in $\mathcal{F}$. By Lemma 5(a), such a node is also in $\mathcal{L}$, and by Lemma 5(b) it is 1-valent

in $\mathcal{L}$. Therefore S'' is not 0-valent in $\mathcal{L}$. By hypothesis, S'' is univalent in $\mathcal{L}$, so it is 1-valent. $\square$

Lemma 7 *Let S_0 and S_1 be two univalent nodes in $\mathcal{L}$ such that the state of some process and each object is the same in $S_0(I)$ as in $S_1(I)$. Then, S_0 and S_1 are either both 0-valent or both 1-valent in $\mathcal{L}$.*

PROOF. Suppose, to the contrary, that S_0 is 0-valent and S_1 is 1-valent. (The other case, where S_0 is 1-valent and S_1 is 0-valent, is symmetric.) Let P be the process that has the same state in $S_0(I)$ and $S_1(I)$.

By Lemma 1, there is a solo schedule S of P such that $S_0 \cdot S$ is a node in $\mathcal{L}$ and P has decided in $S_0 \cdot S(I)$. Since S_0 is 0-valent, P must have decided 0 in $S_0 \cdot S(I)$. Clearly, S_0, S_1 and $S_0 \cdot S$ are all in $\mathcal{F}$. By hypothesis, the state of P and each object is the same in $S_0(I)$ as in $S_1(I)$. Since S is applicable to $S_0(I)$, an easy induction on the length of S shows that S is also applicable to $S_1(I)$, and, furthermore, P and every object is in the same state in $S_0 \cdot S(I)$ as in $S_1 \cdot S(I)$. Hence, $S_1 \cdot S$ is a node in $\mathcal{F}$, and P decides 0 in $S_1 \cdot S(I)$. By Lemma 5(e), as S_1 in 1-valent in $\mathcal{L}$, $\mathit{subtree}(S_1, \mathcal{L}) = \mathit{subtree}(S_1, \mathcal{F})$; it follows that $S_1 \cdot S$ is also in $\mathcal{L}$. This, however, contradicts our hypothesis that S_1 is 1-valent in $\mathcal{L}$, since P has decided 0 in $S_1 \cdot S(I)$. $\square$

Lemma 8 *The root of $\mathcal{L}$ is bivalent.*

PROOF. By Lemma 1, for each $k \in \{0,1\}$, there exists a solo schedule S_k of P_k such that $S_\perp \cdot S_k = S_k$ is a node in $\mathcal{L}$ and P_k has decided in $S_k(I)$. Since P_k has initial value k in I, by Validity, P_k must decide k in $S_k(I)$. Hence, by definition, the root of $\mathcal{L}$ is bivalent. $\square$

Since the root of $\mathcal{L}$ is bivalent while every leaf is univalent, there is a critical node E in $\mathcal{L}$ so that:

- All children of E in $\mathcal{L}$ are univalent in $\mathcal{L}$.
- There are steps e_0 and e_1 of processes P_0 and P_1, respectively, that are applicable to $E(I)$ such that $E \cdot e_0$ and $E \cdot e_1$ have opposite valence in $\mathcal{L}$.

Without loss of generality, assume that $E \cdot e_0$ is 0-valent and $E \cdot e_1$ is 1-valent in $\mathcal{L}$. (Otherwise, we rename e_0 and e_1 and apply the same argument.)

Lemma 9 *Steps e_0 and e_1 access the same object $\mathcal{O}$.*

PROOF. Suppose, for contradiction, that e_0 and e_1 access, respectively, objects O_0 and O_1, where $O_0 \neq O_1$. Then, e_1 is applicable to $E \cdot e_0$ and e_0 is applicable to $E \cdot e_1$ and, in fact, $E \cdot e_0 \cdot e_1(I) = E \cdot e_1 \cdot e_0(I)$. Thus $E \cdot e_0 \cdot e_1$ and $E \cdot e_1 \cdot e_0$ are nodes in $\mathcal{F}$. We shall prove that they are also nodes in $\mathcal{L}$, which will yield a contradiction to Lemma 7:

If $E \cdot e_0 \cdot e_1$ and $E \cdot e_1 \cdot e_0$ are in $\mathcal{L}$, they are 0-valent and 1-valent in $\mathcal{L}$, respectively (because they are children of nodes that are, by assumption, 0-valent and 1-valent in $\mathcal{L}$). However, all processes and objects are in the same state in $E \cdot e_0 \cdot e_1(I)$ as in $E \cdot e_1 \cdot e_0(I)$, contrary to Lemma 7.

We now show that $E \cdot e_0 \cdot e_1$ and $E \cdot e_1 \cdot e_0$ are in $\mathcal{L}$. Since $E \cdot e_1$ is 1-valent in $\mathcal{L}$, by Lemma 5(e), $subtree(E \cdot e_1, \mathcal{L}) = subtree(E \cdot e_1, \mathcal{F})$. Since $E \cdot e_1 \cdot e_0$ is in $\mathcal{F}$, it follows that it is also in $\mathcal{L}$. To show that $E \cdot e_0 \cdot e_1$ is in $\mathcal{L}$, we consider two cases, depending on whether $children(P_1, E, \mathcal{F})$ has one or two nodes.

Case 1. $|children(P_1, E, \mathcal{F})| = 1$: Then that unique child has to be $E \cdot e_1$ (since $E \cdot e_1$ is a node in $\mathcal{F}$). Since $O_0 \neq O_1$, the states of P_1 and O_1 are in the same state in E as in $E \cdot e_0$. It follows that $children(P_1, E \cdot e_0, \mathcal{F}) = \{E \cdot e_0 \cdot e_1\}$. By Lemma 5(d), $E \cdot e_0 \cdot e_1$ is in $\mathcal{L}$.

Case 2. $|children(P_1, E, \mathcal{F})| = 2$: Recall that all nodes in $children(P_1, E, \mathcal{L})$ are univalent. Recall also that, by assumption, $E \cdot e_1$ is in $children(P_1, E, \mathcal{L})$ and is 1-valent. Since $|children(P_1, E, \mathcal{F})| = 2$, by Lemma 6, $children(P_1, E, \mathcal{F}) = children(P_1, E, \mathcal{L})$ and both nodes in $children(P_1, E, \mathcal{L})$ are actually 1-valent in $\mathcal{L}$. Let the two nodes in $children(P_1, E, \mathcal{L})$ be $E \cdot e_1$ and $E \cdot e_1'$. Because P_1 and O_1 are in the same state in E as in $E \cdot e_0$, it must be that $E \cdot e_0 \cdot e_1$ or $E \cdot e_0 \cdot e_1'$ is in $\mathcal{L}$. Since $E \cdot e_1$ and $E \cdot e_1'$ are 1-valent in $\mathcal{L}$ and since e_1 was chosen arbitrarily as a 1-valent node in $children(P_1, E, \mathcal{L})$, we may assume without loss of generality that $E \cdot e_0 \cdot e_1$ is in $\mathcal{L}$. (Otherwise, we repeat the argument with e_1' instead of e_1). □

Lemma 10 *$\mathcal{O}$, the object accessed in e_0 and e_1, is not a register.*

Proof. Assume, by way of contradiction, that $\mathcal{O}$ is a register. Since register is a deterministic type, $children(P_0, E, \mathcal{F})$ and $children(P_1, E, \mathcal{F})$ contain a single node each. Let op_0 and op_1 be the (READ or WRITE) operations applied to $\mathcal{O}$ in steps e_0 and e_1, respectively. There are two cases:

Case 1. *op_k is a* WRITE *operation, for some $k \in \{0, 1\}$:* Let v be the value written into $\mathcal{O}$ by process P_k in step e_k. Since P_k has the same state in $E(I)$ as in $E \cdot e_{\overline{k}}(I)$, e_k is also applicable to $E \cdot e_{\overline{k}}(I)$ and therefore, $children(P_k, E \cdot e_{\overline{k}}, \mathcal{F}) = \{E \cdot e_{\overline{k}} \cdot e_k\}$. By Lemma 5(d), $E \cdot e_{\overline{k}} \cdot e_k$ is in $\mathcal{L}$. Because e_0 and e_1 both access only $\mathcal{O}$, $E \cdot e_k(I)$ and $E \cdot e_{\overline{k}} \cdot e_k(I)$ may differ only in the state of $\mathcal{O}$. Clearly, $\mathcal{O}$ has value v in both $E \cdot e_k(I)$ and $E \cdot e_{\overline{k}} \cdot e_k(I)$. But then the state of process P_k and each object in $E \cdot e_k(I)$ is the same as that in $E \cdot e_{\overline{k}} \cdot e_k(I)$. This contradicts Lemma 7, since $E \cdot e_k$ is k-valent in $\mathcal{L}$ while $E \cdot e_{\overline{k}} \cdot e_k$ is $\overline{k}$-valent (because $E \cdot e_{\overline{k}}$ is) in $\mathcal{L}$.

Case 2. *op_0 and op_1 are both* READ *operations:* Let v be the value of $\mathcal{O}$ in configuration $E(I)$. Since the value of a register is not changed by READ operations, $\mathcal{O}$ has the same value v in $E \cdot e_0(I)$ and $E \cdot e_1(I)$. It follows that e_0 and e_1 are applicable to $E \cdot e_1(I)$ and $E \cdot e_0(I)$, respectively. Thus $children(P_0, E \cdot e_0, \mathcal{F}) = \{E \cdot e_0 \cdot e_1\}$ and *children*$(P_1, E\cdot e_1, \mathcal{F}) = \{E\cdot e_1\cdot e_0\}$. By Lemma 5(d), $E\cdot e_0\cdot e_1$ and $E\cdot e_1\cdot e_0$ are nodes in $\mathcal{L}$ as well. Clearly, the state of each process and each object in $E\cdot e_0\cdot e_1(I)$ is the same as that in $E\cdot e_1\cdot e_0(I)$. This contradicts Lemma 7, since $E\cdot e_0\cdot e_1$ is 0-valent (because $E\cdot e_0$ is) in $\mathcal{L}$ while $E\cdot e_1\cdot e_0$ is 1-valent (because $E\cdot e_1$ is) in $\mathcal{L}$. □

We say that a negation object O is *fresh* (respectively, *upset*) in configuration C if the state of O in C is ☺ (respectively, ☹).

Lemma 11 *$\mathcal{O}$, the object accessed in e_0 and e_1, is not a* negation *object.*

PROOF. Suppose, for contradiction, that $\mathcal{O}$ is a negation object. We consider the following two cases, depending on whether $\mathcal{O}$ is fresh in $E(I)$ or not.

CASE 1. *$\mathcal{O}$ is* not *fresh in $E(I)$:* Then, the state of $\mathcal{O}$ in $E(I)$ is either ☹ or of the form $\langle v, u\rangle$. In both cases, $\mathcal{O}$ is upset in $E\cdot e_0(I)$ and $E\cdot e_1(I)$. Since $\mathcal{O}$ is upset in $E\cdot e_1(I)$, step e_0 is applicable to $E\cdot e_1(I)$. It follows that $E\cdot e_1\cdot e_0$ is a node in $\mathcal{F}$. Since $E\cdot e_1$ is 1-valent in $\mathcal{L}$, by Lemma 5(e), $E\cdot e_1\cdot e_0$ is a 1-valent node in $\mathcal{L}$. Clearly, $\mathcal{O}$ is upset in $E\cdot e_1\cdot e_0(I)$, as $\mathcal{O}$ is upset in $E\cdot e_1(I)$. Thus, the state of P_0 and each object is the same in $E\cdot e_0(I)$ as in $E\cdot e_1\cdot e_0(I)$. This, however, contradicts Lemma 7, since $E\cdot e_0$ is 0-valent in $\mathcal{L}$ while $E\cdot e_1\cdot e_0$ is 1-valent (because $E\cdot e_1$ is) in $\mathcal{L}$.

CASE 2. *$\mathcal{O}$ is fresh in $E(I)$:* For each $k\in\{0,1\}$, let the pending operation of P_k to $\mathcal{O}$ in $E(I)$ be NEGATE(i_k), for some $i_k\in\{0,1,2\}$. Since $\mathcal{O}$ is fresh in $E(I)$, NEGATE(i_k) to $\mathcal{O}$ can return either 0 or 1. Let d_k^0 and d_k^1 be the two steps of P_k that are applicable to $E(I)$, in which $\mathcal{O}$ returns 0 and 1, respectively, to the NEGATE(i_k) operation of P_k. Hence, *children*$(P_k, E, \mathcal{F}) = \{E\cdot d_k^0, E\cdot d_k^1\}$, for $k\in\{0,1\}$. Recall that *children*$(P_1, E, \mathcal{L})$ are univalent, and one of them is 1-valent. By Lemma 6, *children*$(P_1, E, \mathcal{L}) = \{E\cdot d_1^0, E\cdot d_1^1\}$ and both of these nodes are 1-valent in $\mathcal{L}$. Since $E\cdot e_0\in$ *children*$(P_0, E, \mathcal{L})\subseteq$ *children*$(P_0, E, \mathcal{F}) = \{E\cdot d_0^0, E\cdot d_0^1\}$, it must be that $e_0 = d_0^u$ for some $u\in\{0,1\}$. There are two subcases, depending on whether the parameters of the pending NEGATE operations of P_0 and P_1 in $E(I)$ are distinct or not.

> SUBCASE 2(A). $i_0\neq i_1$: Since the state of $\mathcal{O}$ in $E\cdot d_1^u(I)$ is $\langle i_1, u\rangle$, by the specification of negation, d_0^u is the only step of P_0 applicable to $E\cdot d_1^u(I)$. Thus *children*$(P_0, E\cdot d_1^u, \mathcal{F}) = \{E\cdot d_1^u\cdot d_0^u\}$. By Lemma 5(d), $E\cdot d_1^u\cdot d_0^u$ is in $\mathcal{L}$. Since $E\cdot d_1^u$ is 1-valent in $\mathcal{L}$, so is $E\cdot d_1^u\cdot d_0^u$. Repeating the argument above with the role of 0 and 1 interchanged, we can show that $E\cdot d_0^u\cdot d_1^u$ is a 0-valent node in $\mathcal{L}$.
>
> We have shown that $E\cdot d_0^u\cdot d_1^u$ and $E\cdot d_1^u\cdot d_0^u$ are 0-valent and 1-valent in $\mathcal{L}$, respectively. It is easy to verify that all processes and objects have the same state in $E\cdot d_0^u\cdot d_1^u(I)$ as in $E\cdot d_1^u\cdot d_0^u(I)$, contradicting Lemma 7.
>
> SUBCASE 2(B). $i_0\neq i_1$: Similar to the above subcase, replacing d_1^u by $d_1^{\overline{u}}$ throughout.

Since every case leads to contradiction, $\mathcal{O}$ is not a negation object. □

Lemmata 10 and 11 contradict the assumption that $\mathbf{A}_c$ uses only registers and negation objects. Thus,

Theorem 3 *Type* negation *has Consensus number one.*

8 Type **booster** has Consensus number one

In this section, we show that type booster has Consensus number one. Unfortunately, even the modified version of the bivalence argument of Section 7 does not appear to be helpful in proving that Consensus between two processes cannot be solved using only booster objects and registers. Instead, we use a different technique, originally used (in a somewhat different context) in [12]. We briefly describe the proof strategy: Suppose, for contradiction, that the Equality Negation problem for two processes is solvable using only booster objects and registers. Let $\mathbf{A}_{en}$ be an Equality Negation algorithm for two processes that uses a minimal number of booster objects (together with any number of registers). By Theorem 1, $\mathbf{A}_{en}$ must use at least one booster object. We will show that $\mathbf{A}_{en}$ can be modified into an Equality Negation algorithm that uses one booster object less than $\mathbf{A}_{en}$, thus contradicting the definition of $\mathbf{A}_{en}$.

8.1 Valence of nodes in Equality Negation computation trees

To carry out the proof outlined above we use computation trees of algorithm $\mathbf{A}_{en}$. It is necessary to redefine the concept of the valence of a node in a computation tree (defined in Section 7.1) to account for the fact that the algorithm is now solving a different problem, namely Equality Negation rather than Consensus. To motivate the new definition, recall from Section 7.1 that the valence of a node indicates whether the outcome of Consensus in the (perhaps partial) execution represented by the node has been determined (univalent node) or not (bivalent node); and if the outcome has been determined, what it is. In the case of Consensus, it was possible to identify the outcome with the decision of the processes. In the case of Equality Negation this is not quite so; in addition to the decision of a process we must also take into account whether the two processes start off with identical or distinct initial values. We therefore identify two possible outcomes of the problem in the following way: If the processes have distinct initial values, then outcome 0 is when both processes decide 0, and outcome 1 is when both processes decide 1 — just as in Consensus. If, however, processes have the same initial value, we (arbitrarily) designate outcome 0 as that in which P_k decides k (for all $k \in \{0,1\}$), and outcome 1 as that in which P_k decides $\overline{k}$ (for all $k \in \{0,1\}$).

With this revised notion of the "outcome" for Equality Negation in mind, valence is now defined as follows. Let I be an initial configuration of algorithm $\mathbf{A}_{en}$, and $\mathcal{T}$ be a computation tree of $\mathbf{A}_{en}$ from I. There are two cases:

Case 1. *The two processes have different initial values in I:* S is *bivalent* in $\mathcal{T}$ if S has descendents S_0 and S_1, so that some process decides 0 in $S_0(I)$ and some process decides

1 in $S_1(I)$. S is v-valent (for $v \in \{0,1\}$) in $\mathcal{T}$ if for any descendent S' of S, no process decides $\overline{v}$ in $S'(I)$.

CASE 2. *The two processes have the same initial value in I:* S is *bivalent* in $\mathcal{T}$ if S has descendents S_0 and S_1, so that in $S_0(I)$ P_0 decides 0 or P_1 decides 1, while in $S_1(I)$ P_0 decides 1 or P_1 decides 0. S is v-valent (for $v \in \{0,1\}$) in $\mathcal{T}$ if for any descendent S' of S, P_0 does not decide $\overline{v}$ and P_1 does not decide v in $S'(I)$.

As before, S is *univalent* if it is either 0-valent or 1-valent. It is easy to verify, directly from the Termination and Negation properties, that each node in $\mathcal{T}$ is either univalent or bivalent, and each leaf is univalent.

We now prove some simple consequences of this definition. Let $\mathbf{A}_{en}$ be an Equality Negation algorithm for processes P_0 and P_1, I be an initial configuration of $\mathbf{A}_{en}$, and $\mathcal{F}$ be the full computation tree of $\mathbf{A}_{en}$ from I.

Lemma 12 *Let S_0 and S_1 be two univalent nodes in $\mathcal{F}$ such that some process has decided the same value in both $S_0(I)$ and $S_1(I)$. Then, S_0 and S_1 are either both 0-valent or both 1-valent in $\mathcal{F}$.*

PROOF. Let P_k, for some $k \in \{0,1\}$, be the process that has decided the same value, say v, in both $S_0(I)$ and $S_1(I)$. There are two cases:

CASE 1. *P_0 and P_1 have different initial values in I:* Then both S_0 and S_1 are v-valent in $\mathcal{F}$.

CASE 2. *P_0 and P_1 have the same initial value in I:* If $v = k$, then both S_0 and S_1 are 0-valent in $\mathcal{F}$. If $v = \overline{k}$, then both S_0 and S_1 are 1-valent in $\mathcal{F}$. $\square$

Lemma 13 *Let S_0 and S_1 be two univalent nodes in $\mathcal{F}$ such that the state of some process and each object is the same in $S_0(I)$ as in $S_1(I)$. Then, S_0 and S_1 are either both 0-valent or both 1-valent in $\mathcal{F}$.*

PROOF. Let P be the process that has the same state in $S_0(I)$ as in $S_1(I)$. Since S_0 is a node in $\mathcal{F}$, by Lemma 1, there exists a solo schedule S of P such that $S_0 \cdot S$ is a node in $\mathcal{F}$ and P has decided in $S_0 \cdot S(I)$. Using the fact the the state of process P and each object in $S_0(I)$ is the same as that in $S_1(I)$, we can show, by induction on the length of S, that S is also applicable to $S_1(I)$. Thus, $S_1 \cdot S$ is a node in $\mathcal{F}$ (because $\mathcal{F}$ is a full computation tree). P decides the same value in $S_0 \cdot S(I)$ as in $S_1 \cdot S(I)$, since it is in the same state in both configurations. By Lemma 12, $S_0 \cdot S$ and $S_1 \cdot S$ are either both 0-valent or both 1-valent in $\mathcal{F}$. $\square$

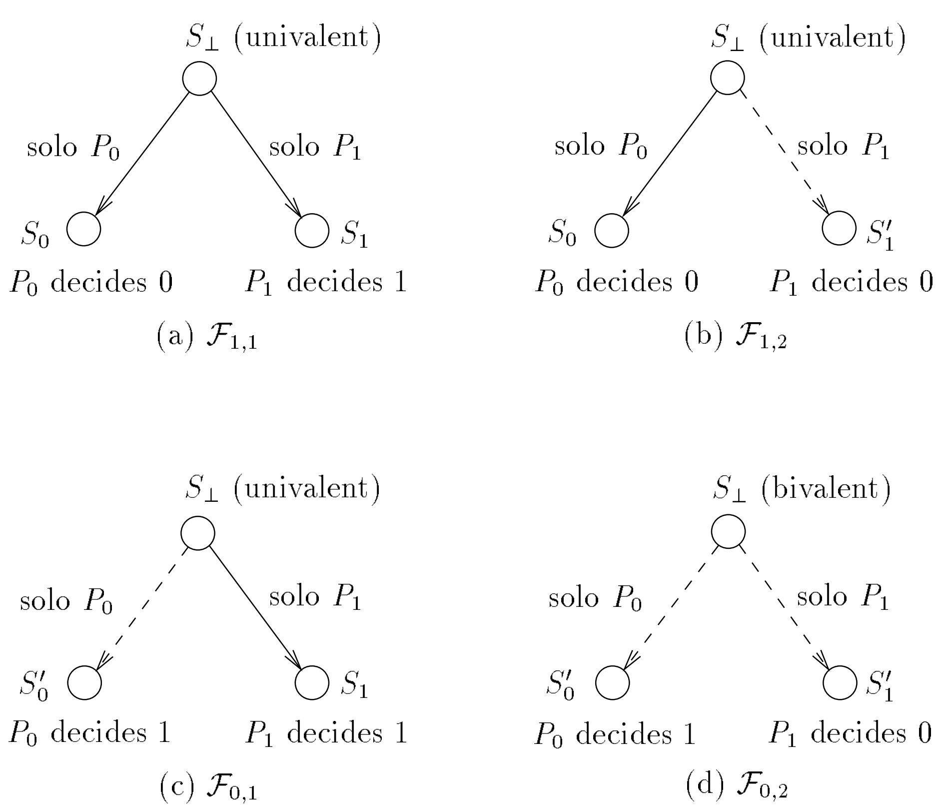

Figure 9: Full computation trees of $\mathbf{A}_{en}$

8.2 The impossibility result

In this section we deal exclusively with *full* computation trees of $\mathbf{A}_{en}$. Recall that, once we have fixed the algorithm and its initial configuration, the full computation tree is unique — even if the algorithm uses nondeterministic types. For any $i, j \in \{0, 1, 2\}$, let $I_{i,j}$ be the initial configuration of $\mathbf{A}_{en}$ in which P_0 and P_1 have initial values i and j, respectively, and let $\mathcal{F}_{i,j}$ be the full computation tree of $\mathbf{A}_{en}$ from $I_{i,j}$.

Lemma 14 *For some initial configuration I of algorithm $\mathbf{A}_{en}$, the full computation tree $\mathcal{F}$ of $\mathbf{A}_{en}$ from I has a bivalent root.*

Proof. Suppose, for contradiction, that the root of every full computation tree of $\mathbf{A}_{en}$ is univalent. Since the root of $\mathcal{F}_{1,1}$ is univalent, the nodes in $\mathcal{F}_{1,1}$ are either all 0-valent or all 1-valent. Without loss of generality, assume that they are 0-valent. From Lemma 1 it follows that, for each $k \in \{0, 1\}$, there is a solo schedule S_k of P_k such that $S_\perp \cdot S_k = S_k$ is a node in $\mathcal{F}$ and P_k has decided in $S_k(I_{1,1})$. Since S_k is 0-valent in $\mathcal{F}_{1,1}$, the definition of valence implies that P_k decides k in $S_k(I_{1,1})$; see Figure 9(a).

Since $I_{1,1}$ and $I_{1,2}$ differ only in the state of process P_1 and since S_0 is a solo schedule of P_0, a straightforward induction on the length of S_0 shows that S_0 is applicable to $I_{1,2}$. This implies that S_0 is node in $\mathcal{F}_{1,2}$ and P_0 decides 0 in $S_0(I_{1,2})$ (because P_0 has the same state in $S_0(I_{1,2})$ as in $S_0(I_{1,1})$); see Figure 9(b). By definition of valence and the hypothesis that the root of $\mathcal{F}_{1,2}$ is univalent, it follows that the root of $\mathcal{F}_{1,2}$ is 0-valent. By applying Lemma 1 to the root of $\mathcal{F}_{1,2}$, there is a solo schedule S'_1 of P_1 such that $S_\perp \cdot S'_1 = S'_1$ is a node in $\mathcal{F}_{1,2}$ and P_1 has decided in $S'_1(I_{1,2})$. P_1 decides 0 in $S'_1(I_{1,2})$, because the root of $\mathcal{F}_{1,2}$ is 0-valent and $\mathcal{F}_{1,2}$ is a computation tree from an initial configuration where processes have different initial values. Repeating the argument above with $I_{1,2}$ replaced by $I_{0,1}$ and with the role of 0 and 1 interchanged, we can prove that there exists a solo schedule S'_0 of P_0 such that S'_0 is a node in $\mathcal{F}_{0,1}$ and P_0 decides 1 in $S'_0(I_{0,1})$; see Figure 9(c).

By induction on the length of S'_0 and S'_1 we can show that S'_0 and S'_1 are applicable to $I_{0,2}$. Hence, both S'_0 and S'_1 are nodes in $\mathcal{F}_{0,2}$; see Figure 9(d). P_0 decides 1 in $S'_0(I_{0,2})$, because P_0 has the same state in $S'_0(I_{0,2})$ as in $S'_0(I_{0,1})$. Similarly, P_1 decides 0 in $S'_1(I_{0,2})$, because P_1 has the same state in $S'_0(I_{0,2})$ as in $S'_0(I_{1,2})$. Thus, the root of $\mathcal{F}_{0,2}$ has descendents, namely, S'_0 and S'_1, in which processes have decided different values in the corresponding configurations. Since $\mathcal{F}_{0,2}$ is a computation tree from an initial configuration where the processes have different initial values, this means that the root of $\mathcal{F}_{0,2}$ is bivalent. This contradicts our hypothesis that the root of every full computation tree of $\mathbf{A}_{en}$ is univalent. $\square$

In the rest of the section we focus exclusively on the initial configuration I and the full computation tree $\mathcal{F}$ (of $\mathbf{A}_{en}$ from I) whose existence is asserted by Lemma 14. Thus, from now on, we refer to the valence of a node without explicitly specifying the computation tree, since that will always be $\mathcal{F}$.

Since the root of $\mathcal{F}$ is bivalent and every leaf is univalent, there is a critical node E such that:

- All children of E in $\mathcal{F}$ are univalent.
- There are steps e_0 and e_1 of processes P_0 and P_1, respectively, that are applicable to $E(I)$ such that $E \cdot e_0$ and $E \cdot e_1$ have opposite valence.

Without loss of generality, assume that $E \cdot e_0$ is 0-valent and $E \cdot e_1$ is 1-valent (if not, we simply rename e_0 and e_1). Using arguments similar to that in Lemmata 9 and 10,[5] we can show that e_0 and e_1 access the *same* object $\mathcal{O}$, and that object cannot be a register; thus, $\mathcal{O}$ is a booster object. We say that $\mathcal{O}$ is *fresh* (respectively, *upset*) in a configuration C if the state of $\mathcal{O}$ in C is ☺ (respectively, ☹).

Lemma 15 *Object $\mathcal{O}$ is not upset in $E \cdot e_0(I)$ or in $E \cdot e_1(I)$.*

[5]except simpler, since we are now working with the full computation tree and need not worry about whether certain schedules applicable to I have been pruned.

PROOF. Suppose, for contradiction, that $\mathcal{O}$ is upset in $E \cdot e_k(I)$, for some $k \in \{0,1\}$. By Lemma 1 (applied to node $E \cdot e_{\overline{k}}$), there is a solo schedule $S_{\overline{k}}$ of $P_{\overline{k}}$ such that $E \cdot e_{\overline{k}} \cdot S_{\overline{k}}$ is a node in $\mathcal{F}$ and $P_{\overline{k}}$ has decided in $E \cdot e_{\overline{k}} \cdot S_{\overline{k}}(I)$. Comparing configurations $E(I)$ and $E \cdot e_k(I)$, we note that $P_{\overline{k}}$ and each object other than $\mathcal{O}$ is in the same state in both. Using this and the fact that $\mathcal{O}$ is upset in $E \cdot e_k(I)$, a straightforward induction on the length of $S_{\overline{k}}$ shows that $e_{\overline{k}} \cdot S_{\overline{k}}$ is applicable to $E \cdot e_k(I)$. Thus, $E \cdot e_k \cdot e_{\overline{k}} \cdot S_{\overline{k}}$ is a node in $\mathcal{F}$, and $P_{\overline{k}}$ decides the same value in $E \cdot e_k \cdot e_{\overline{k}} \cdot S_{\overline{k}}(I)$ as in $E \cdot e_{\overline{k}} \cdot S_{\overline{k}}(I)$. By Lemma 12, $E \cdot e_k \cdot e_{\overline{k}} \cdot S_{\overline{k}}$ and $E \cdot e_{\overline{k}} \cdot S_{\overline{k}}$ have the same valence. Since $E \cdot e_k$ and $E \cdot e_{\overline{k}}$ are univalent and have descendents of the same valence, they also have the same valence. This contradicts the fact that $E \cdot e_0$ is 0-valent while $E \cdot e_1$ is 1-valent. $\square$

Lemma 16 *The state of $\mathcal{O}$ in $E(I)$ is not of the form $\langle v_0, v_1, u, d \rangle$.*

PROOF. Let op_0 and op_1 be the pending (ENROLL or REVEAL) operations to $\mathcal{O}$ of processes P_0 and P_1 in $E(I)$, respectively. First we show that e_0 and e_1 are applicable to $E \cdot e_1(I)$ and $E \cdot e_0(I)$, respectively, and the state of $\mathcal{O}$ in $E \cdot e_0 \cdot e_1(I)$ is the same as that in $E \cdot e_1 \cdot e_0(I)$. There are four cases:

CASE 1. *$op_0 =$ ENROLL(i_0) and $op_1 =$ ENROLL(i_1), for some $i_0, i_1 \in \{0,1\}$:* Note that an ENROLL operation to a booster object, in any state, can return any value in $\{0,1,2\}$. Thus, e_0 and e_1 are applicable to $E \cdot e_1(I)$ and $E \cdot e_0(I)$, respectively. We shall show that $\mathcal{O}$ is upset in both $E \cdot e_1 \cdot e_0(I)$ and $E \cdot e_0 \cdot e_1(I)$.

It is impossible that both $v_0 = \perp$ and $v_1 = \perp$ (otherwise, by the specification of booster, $\mathcal{O}$ is upset in $E \cdot e_k(I)$, for each $k \in \{0,1\}$, contrary to Lemma 15). Without loss of generality, assume that $v_0 \neq \perp$. Then, it must be that $i_0 = i_1 = 1$ (otherwise, $i_k = 0$, for some $k \in \{0,1\}$, and by the specification of booster, $\mathcal{O}$ will be upset in $E \cdot e_k(I)$, contrary to Lemma 15). Furthermore, we must have $v_1 = \perp$ (otherwise $\mathcal{O}$ will be upset in both $E \cdot e_0(I)$ and $E \cdot e_1(I)$, which again contradicts Lemma 15). Since $i_0 = i_1 = 1$ and $v_1 = \perp$, the state of $\mathcal{O}$ in $E \cdot e_0(I)$ is $\langle v_0, r_0, u, d \rangle$ while in $E \cdot e_1(I)$ is $\langle v_0, r_1, u, d \rangle$, where r_k is the response of $\mathcal{O}$ in step e_k, for each $k \in \{0,1\}$. Note that $r_k \neq \perp$, for each $k \in \{0,1\}$. By the specification of booster, an ENROLL(1) operation to a booster object in a state of the form $\langle *, r, *, * \rangle$ with $r \neq \perp$ will upset the object.[6] Therefore, $\mathcal{O}$ is upset in both $E \cdot e_1 \cdot e_0(I)$ and $E \cdot e_0 \cdot e_1(I)$, as wanted.

CASE 2. *$op_0 =$ ENROLL(i_0) and $op_1 =$ REVEAL(i_1), for some $i_0, i_1 \in \{0,1\}$:* Since e_0 is an ENROLL step to $\mathcal{O}$, as argued in Case 1, e_0 is applicable to $E \cdot e_1(I)$. We now show that e_1 is applicable to $E \cdot e_0(I)$. To simplify the presentation assume, without loss of generality, that $i_0 = 0$. Since e_0 is an ENROLL(0) to $\mathcal{O}$, we must have $v_0 = \perp$. (If $v_0 \neq \perp$, then $\mathcal{O}$ will be upset in $E \cdot e_0(I)$ because an ENROLL(0) to a booster object in a state of the form $\langle v, *, *, * \rangle$ with $v \neq \perp$ will upset the object, contrary to Lemma 15.) Again by Lemma 15, as e_1 is a REVEAL(i_1) to $\mathcal{O}$, it must be that $u = \perp$. (If $u \neq \perp$, then $\mathcal{O}$ will be

[6] We use $*$ to indicate "don't care" entries in a tuple.

upset in $E \cdot e_1(I)$, because a REVEAL to a booster object in a state of the form $\langle *, *, w, * \rangle$ with $w \neq \perp$ will also upset the object, which contradicts Lemma 15.) Thus, the state of $\mathcal{O}$ in $E(I)$ is $\langle \perp, v_1, \perp, d \rangle$; and the state of $\mathcal{O}$ in $E \cdot e_0(I)$ is $\langle r_0, v_1, \perp, d \rangle$, where r_0 is the response that P_0 receives when it applies ENROLL(0) to $\mathcal{O}$ (i.e., the response of step e_0). By the specification of booster, REVEAL(i_1) to a booster object in a state of the form $\langle *, *, \perp, d \rangle$ must return d. Therefore, $(P_1, \text{REVEAL}(i_1), \mathcal{O}, d)$ is the only step of P_1 that is applicable to $E(I)$ and $E \cdot e_0(I)$. Since, by assumption, e_1 is a step of P_1 that is applicable to $E(I)$, it follows that $e_1 = (P_1, \text{REVEAL}(i_1), \mathcal{O}, d)$. Thus, e_1 is applicable to $E \cdot e_0(I)$.

We have shown that e_0 and e_1 are applicable to $E \cdot e_1(I)$ and $E \cdot e_0(I)$, respectively. From the specification of booster, it is easy to verify that $\mathcal{O}$ has the same state, namely $\langle r_0, v_1, i_1, d \rangle$, in $E \cdot e_1 \cdot e_0(I)$ as in $E \cdot e_0 \cdot e_1(I)$.

CASE 3. *op_0 = REVEAL(i_0) and op_1 = ENROLL(i_1), for some $i_0, i_1 \in \{0, 1\}$:* Similar to Case 2.

CASE 4. *op_0 = REVEAL(i_0) and op_1 = REVEAL(i_1), for some $i_0, i_1 \in \{0, 1\}$:* Since both e_0 and e_1 are REVEAL steps to $\mathcal{O}$, using an argument similar to that in Case 2, we can show that $u = \perp$, $e_0 = (P_0, \text{REVEAL}(i_0), \mathcal{O}, d)$ and $e_1 = (P_1, \text{REVEAL}(i_1), \mathcal{O}, d)$. Hence, $\mathcal{O}$ is in state $\langle v_0, v_1, i_k, d \rangle$ in $E \cdot e_k(I)$, for each $k \in \{0, 1\}$. By the specification of booster, a REVEAL operation to a booster object in state $\langle v_0, v_1, i_k, d \rangle$, for $i_k \neq \perp$, will upset the object and return either 0 or 1. Therefore, e_0 and e_1 are applicable to $E \cdot e_1(I)$ and $E \cdot e_0(I)$, respectively, and $\mathcal{O}$ is upset in both $E \cdot e_1 \cdot e_0(I)$ and $E \cdot e_0 \cdot e_1(I)$.

Since e_0 and e_1 are applicable to $E \cdot e_1(I)$ and $E \cdot e_0(I)$, respectively, both $E \cdot e_1 \cdot e_0$ and $E \cdot e_0 \cdot e_1$ are nodes in $\mathcal{F}$. Since e_0 and e_1 access the same object $\mathcal{O}$, the state of each process and each object other than $\mathcal{O}$ is the same in $E \cdot e_1 \cdot e_0(I)$ as in $E \cdot e_0 \cdot e_1(I)$. But we have just shown that the state of $\mathcal{O}$, too, is the same in $E \cdot e_0 \cdot e_1(I)$ as in $E \cdot e_1 \cdot e_0(I)$. Thus, each process and object is in the same state in $E \cdot e_0 \cdot e_1(I)$ as in $E \cdot e_1 \cdot e_0(I)$. By Lemma 13, $E \cdot e_0 \cdot e_1$ and $E \cdot e_1 \cdot e_0$ have the same valence. Since $E \cdot e_0$ and $E \cdot e_1$ are univalent, $E \cdot e_0$ and $E \cdot e_1$ also have the same valence. This, however, contradicts the fact that $E \cdot e_0$ is 0-valent while $E \cdot e_1$ is 1-valent. □

By Lemmata 15 and 16, $\mathcal{O}$ must be fresh in $E(I)$. Furthermore, each P_k must have an ENROLL(i_k) operation to $\mathcal{O}$ pending in $E(I)$ (otherwise, by the specification of booster, $\mathcal{O}$ would be upset in $E \cdot e_k$, contrary to Lemma 15). Finally, we also have that $i_0 \neq i_1$. (Otherwise, $\mathcal{O}$ is upset in both $E \cdot e_1 \cdot e_0(I)$ and $E \cdot e_0 \cdot e_1(I)$, which in turn implies that every process and object is in the same state in $E \cdot e_0 \cdot e_1(I)$ as in $E \cdot e_1 \cdot e_0(I)$. By Lemma 13, $E \cdot e_0 \cdot e_1$ and $E \cdot e_1 \cdot e_0$ have the same valence. This contradicts the fact that $E \cdot e_k \cdot e_{\overline{k}}$ is k-valent, for each $k \in \{0, 1\}$.) Without loss of generality, assume that $i_0 = 0$ and $i_1 = 1$ (if not, simply exchange the names of the parameters of ENROLL and use the argument below).

To summarise: $\mathcal{O}$ is fresh in $E(I)$, and each P_k has an ENROLL(k) operation to $\mathcal{O}$ pending in $E(I)$. Since $\mathcal{O}$ is fresh in $E(I)$, an ENROLL operation to $\mathcal{O}$ can return 0, 1 or 2. For each $k \in \{0,1\}$, and $v \in \{0,1,2\}$, let $e_k^v = (P_k, \text{ENROLL}(k), \mathcal{O}, v)$ — i.e., the step in which P_k applies ENROLL(k) to $\mathcal{O}$ and receives v as the response. Thus, $children(P_k, E, \mathcal{F}) = \{E \cdot e_k^v : v \in \{0,1,2\}\}$.

Lemma 17 *For each $k \in \{0,1\}$, the nodes in $children(P_k, E, \mathcal{F})$ are either all 0-valent or all 1-valent.*

PROOF. We prove the lemma for $k = 0$. (The proof for $k = 1$ can be obtained by interchanging the role of 0 and 1 in the argument below.) Recall that all the nodes in $children(P_0, E, \mathcal{F})$ are univalent. To establish the lemma, it suffices to show that for any $v, v' \in \{0,1,2\}$, $E \cdot e_0^v$ and $E \cdot e_0^{v'}$ have the same valence. From the specification of booster and the fact that $\mathcal{O}$ is fresh in $E(I)$ we have:

(a) For any $v \in \{0,1,2\}$, the state of $\mathcal{O}$ in $E \cdot e_0^v(I)$ is $\langle v, \perp, \perp, 0\rangle$.

(b) For any $v \in \{0,1,2\}$, if S_1 is a solo schedule of P_1 applicable to $E \cdot e_k^v(I)$, then the state of $\mathcal{O}$ in $E \cdot e_k^v \cdot S_1(I)$ is either of the form $\langle *, *, *, 0\rangle$ or ☹.

(c) A REVEAL operation can return 0 when applied to a booster object in a state of the form $\langle *, *, *, 0\rangle$ or in state ☹.

Let v be any value in $\{0,1,2\}$. By Lemma 1, there is a solo schedule S_1 of P_1 such that $E \cdot e_0^v \cdot S_1$ is a node in $\mathcal{F}$ and P_1 has decided in $E \cdot e_0^v \cdot S_1(I)$. Since $\mathcal{F}$ is a full computation tree, by (a)–(c) we may assume, without loss of generality, that S_1 contains *no* step of the form $(P_1, \text{REVEAL}(i), \mathcal{O}, 1)$.

For any $v' \in \{0,1,2\}$, P_1 and each object except $\mathcal{O}$ is in the same state in $E \cdot e_0^v(I)$ as in $E \cdot e_0^{v'}(I)$. By induction on the length of S_1, and using properties (a)–(c), it can be shown that S_1 is applicable to $E \cdot e_0^{v'}(I)$, and that P_1 and each object except $\mathcal{O}$ is in the same state in $E \cdot e_0^v \cdot S_1(I)$ as in $E \cdot e_0^{v'} \cdot S_1(I)$. Hence P_1 decides the same value in both configurations. By Lemma 12, nodes $E \cdot e_0^v \cdot S_1$ and $E \cdot e_0^{v'} \cdot S_1$ have the same valence. Hence, $E \cdot e_0^v$ and $E \cdot e_0^{v'}$ (which, by assumption, are univalent), must also have the same valence, as wanted. ☐

Lemma 18 *For any node $E \cdot S$ in $\mathcal{F}$, if P_k is the process that takes the first step in S, then $E \cdot S$ is k-valent.*

PROOF. Let P_k be the process that takes the first step in S. Since $children(P_k, E, \mathcal{F}) = \{E \cdot e_k^v : v \in \{0,1,2\}\}$, the first step of S must be e_k^v, for some $v \in \{0,1,2\}$. By Lemma 17, all nodes in $children(P_k, E, \mathcal{F})$ have the same valence. One of these children, namely $E \cdot e_k$ is k-valent. Therefore, $E \cdot e_k^v$ is k-valent. Since $E \cdot S$ is a descendent of $E \cdot e_k^v$, it follows that $E \cdot S$ is k-valent. ☐

Lemma 19 *Let $E \cdot S$ be any node in $\mathcal{F}$, and $k \in \{0,1\}$. The first step in S taken by P_k, if any, is an* ENROLL(k) *to $\mathcal{O}$.*

PROOF. Immediate from the fact that each process P_k has an ENROLL(k) operation to $\mathcal{O}$ pending in $E(I)$. $\square$

Lemma 20 *Let $E \cdot S$ be any node in $\mathcal{F}$ such that S contains no* REVEAL *step to $\mathcal{O}$, and d_k be the first step of P_k in S, for any $k \in \{0,1\}$. Let $\hat{S}$ be the schedule obtained from S by moving d_k to the beginning (i.e., if $S = S' \cdot d_k \cdot S''$ for some S' and S'', then $\hat{S} = d_k \cdot S' \cdot S''$). Then, $E \cdot \hat{S}$ is a node in $\mathcal{F}$, and the state of each process and each object other than $\mathcal{O}$ is the same in $E \cdot \hat{S}(I)$ as in $E \cdot S(I)$.*

PROOF. Let S' and S'' be such that $S = S' \cdot d_k \cdot S''$. Since d_k is the first step of P_k in S, S' is a solo schedule of $P_{\overline{k}}$ and, by Lemma 19, d_k is an ENROLL to $\mathcal{O}$. By the specification of booster, d_k is applicable to $E(I)$. Because d_k is a step of P_k that accesses (only) $\mathcal{O}$, the state of $P_{\overline{k}}$ and each object other than $\mathcal{O}$ is the same in $E(I)$ as in $E \cdot d_k(I)$. Since S' is a solo schedule of $P_{\overline{k}}$ and by hypothesis does not contain a REVEAL step to $\mathcal{O}$, we can show, by induction on the length of S', that S' is also applicable to $E \cdot d_k(I)$. Thus $E \cdot d_k \cdot S'$ is a node in $\mathcal{F}$. It is easy to verify that the state of each process and each object other than $\mathcal{O}$ is the same in $E \cdot d_k \cdot S'(I)$ as in $E \cdot S' \cdot d_k(I)$. By hypothesis, S'' is applicable to $E \cdot S' \cdot d_k(I)$ and does not contain a REVEAL step to $\mathcal{O}$. By induction on the length of S'', we can show that S'' is applicable to $E \cdot d_k \cdot S'(I)$. Hence, $E \cdot d_k \cdot S' \cdot S''$ is a node in $\mathcal{F}$, and the state of each process and each object other than $\mathcal{O}$ is the same in $E \cdot d_k \cdot S' \cdot S''(I)$ as in $E \cdot S' \cdot d_k \cdot S''(I)$. $\square$

Lemma 21 *Let $E \cdot S$ be any node in $\mathcal{F}$ such that some process has decided in $E \cdot S(I)$. Then, S has a* REVEAL *step to $\mathcal{O}$.*

PROOF. Suppose, for contradiction, that some process has decided in $E \cdot S(I)$, but S contains no REVEAL step to $\mathcal{O}$. Without loss of generality, we may assume that both processes take steps in S. (If S contains only steps of process P_k for some $k \in \{0,1\}$, we extend S by a step of $P_{\overline{k}}$, where by Lemma 19 that step of $P_{\overline{k}}$ cannot be a REVEAL to $\mathcal{O}$, and consider the extended schedule instead.)

Let P_ℓ, $\ell \in \{0,1\}$, be the process that does *not* take the first step in S, and $\hat{S}$ be the schedule obtained from S by moving the first step of P_ℓ to the beginning. By Lemma 20, $E \cdot \hat{S}$ is a node in $\mathcal{F}$, and the state of each process and each object except $\mathcal{O}$ is the same in $E \cdot \hat{S}(I)$ as in $E \cdot S(I)$. In particular, the process that has decided in $E \cdot S(I)$ has decided the same value in $E \cdot \hat{S}(I)$. By Lemma 12, $E \cdot S$ and $E \cdot \hat{S}$ have the same valence. This contradicts Lemma 18, since P_ℓ takes the first step in $\hat{S}$ while $P_{\overline{\ell}}$ takes the first step in S. $\square$

Lemma 22 *Let $E \cdot S$ be any node in $\mathcal{F}$ such that S contains no* REVEAL *step to $\mathcal{O}$. Then, S contains at most one* ENROLL *step to $\mathcal{O}$ by each process.*

PROOF. Suppose, for contradiction, that S has two ENROLL steps to $\mathcal{O}$ by some process, say P_k, for some $k \in \{0,1\}$. As in the proof of Lemma 21 we may assume, without loss of generality, that both processes have taken steps in S. By Lemma 19, the first step taken by $P_{\overline{k}}$ in S is an ENROLL to $\mathcal{O}$. Together with the assumed two ENROLL steps to $\mathcal{O}$ by P_k, S has at least three ENROLL steps to $\mathcal{O}$.

Let P_ℓ, $\ell \in \{0,1\}$, be the process that does *not* take the first step in S. Let $\hat{S}$ be the schedule obtained from S by moving the first of step of P_ℓ to the beginning. By Lemma 20, $E \cdot \hat{S}$ is a node in $\mathcal{F}$, and each process and each object except $\mathcal{O}$ is in the same state in $E \cdot S(I)$ as in $E \cdot \hat{S}(I)$. Since S and $\hat{S}$ each contains at least three ENROLL steps to $\mathcal{O}$, it can be easily verified from the specification of booster that $\mathcal{O}$ is upset in both $E \cdot S(I)$ and $E \cdot \hat{S}(I)$. This implies that $E \cdot S(I) = E \cdot \hat{S}(I)$. By Lemma 13, $E \cdot S$ and $E \cdot \hat{S}$ have the same valence. This contradicts Lemma 18, since P_ℓ takes the first step in $\hat{S}$ while $P_{\overline{\ell}}$ takes the first step in S. □

Lemma 23 *Let $E \cdot S$ be any node in $\mathcal{F}$ such that S contains no* REVEAL *step to $\mathcal{O}$ and some process P_k has a* REVEAL *operation to $\mathcal{O}$ pending in $E \cdot S(I)$. Then, S contains exactly one* ENROLL *step to $\mathcal{O}$ by P_k.*

PROOF. Since the operation of P_k pending in $E(I)$ is an ENROLL to $\mathcal{O}$, while in $E \cdot S(I)$ it is a REVEAL to $\mathcal{O}$, S must contain at least one step by P_k. By Lemma 19, the first step taken by P_k in S is an ENROLL to $\mathcal{O}$. By Lemma 22, S cannot contain two such steps. □

Lemma 24 *Let $E \cdot S$ be any node in $\mathcal{F}$ such that S contains no* REVEAL *step to $\mathcal{O}$ and both processes have a* REVEAL *operation to $\mathcal{O}$ pending in $E \cdot S(I)$. For each $k \in \{0,1\}$, let v_k be the value returned by the* ENROLL *step to $\mathcal{O}$ of P_k that is contained in S (this is well-defined by Lemma 23); and i_k be the parameter of the* REVEAL *to $\mathcal{O}$ that is the pending operation of P_k in $E \cdot S(I)$. Then, $v_0 = v_1$ if and only if $i_0 \neq i_1$.*

PROOF. Since both processes have an ENROLL operation to $\mathcal{O}$ pending in $E(I)$, while both have a REVEAL operation to $\mathcal{O}$ pending in $E \cdot S(I)$, both processes must have taken steps in S. Let P_ℓ, $\ell \in \{0,1\}$, be the process that does *not* take the first step in S, and $\hat{S}$ be the schedule obtained from S by moving the first step of P_ℓ to the beginning. By Lemma 20, each process and each object except $\mathcal{O}$ is in the same state in $E \cdot S(I)$ as in $E \cdot \hat{S}(I)$. By hypothesis, for each $k \in \{0,1\}$, P_k has a REVEAL(i_k) to $\mathcal{O}$ pending in $E \cdot S(I)$ — and hence in $E \cdot \hat{S}(I)$. Let $d_{\overline{\ell}}$ and $\hat{d}_{\overline{\ell}}$ be REVEAL$(i_{\overline{\ell}})$ steps of $P_{\overline{\ell}}$ that are applicable to $E \cdot S(I)$ and $E \cdot \hat{S}(I)$, respectively.

Recall that $\mathcal{O}$ is fresh in $E(I)$, and in schedules S and $\hat{S}$ each P_k has applied exactly one operation to $\mathcal{O}$, namely ENROLL(k), obtaining v_k as the response. Given this, it is easy to verify from the specification of booster that the state of $\mathcal{O}$ in both $E \cdot S \cdot d_{\overline{\ell}}(I)$ and $E \cdot \hat{S} \cdot \hat{d}_{\overline{\ell}}(I)$ is of the form $\langle v_0, v_1, i_{\overline{\ell}}, * \rangle$.

Assume, contrary to the lemma, that it is not the case that $v_0 = v_1$ if and only if $i_0 \neq i_1$. By the specification of booster, it is easy to verify that a REVEAL(i_ℓ) operation applied to $\mathcal{O}$ when the state of $\mathcal{O}$ is of the form $\langle v_0, v_1, i_{\overline{\ell}}, *\rangle$ will cause $\mathcal{O}$ to become upset, and can return any response whatever (in verifying this, note that $i_{\overline{\ell}} \neq \perp$). Consider the step $d_\ell = (P_\ell, \text{REVEAL}(i_\ell), \mathcal{O}, r)$ of P_ℓ, where r is any response in $\{0, 1\}$. Note that the operation of P_ℓ that is pending in both $E \cdot S \cdot d_{\overline{\ell}}(I)$ and $E \cdot \hat{S} \cdot \hat{d}_{\overline{\ell}}(I)$ is REVEAL(i_ℓ) to $\mathcal{O}$. Thus, as we just argued, d_ℓ is applicable to both $E \cdot S \cdot d_{\overline{\ell}}(I)$ and $E \cdot \hat{S} \cdot \hat{d}_{\overline{\ell}}(I)$ (since the state of $\mathcal{O}$ in both is of the form $\langle v_0, v_1, i_{\overline{\ell}}, *\rangle$); and $\mathcal{O}$ is upset in both $E \cdot S \cdot d_{\overline{\ell}} \cdot d_\ell(I)$ and $E \cdot \hat{S} \cdot \hat{d}_{\overline{\ell}} \cdot d_\ell(I)$. By definition, both $E \cdot S \cdot d_{\overline{\ell}} \cdot d_\ell$ and $E \cdot \hat{S} \cdot \hat{d}_{\overline{\ell}} \cdot d_\ell$ are nodes in $\mathcal{F}$.

Recall that each process and each object except $\mathcal{O}$ is in the same state in $E \cdot S(I)$ as in $E \cdot \hat{S}(I)$. Thus, P_ℓ and *all* objects (including $\mathcal{O}$) are in the same state in $E \cdot S \cdot d_{\overline{\ell}} \cdot d_\ell(I)$ as in $E \cdot \hat{S} \cdot \hat{d}_{\overline{\ell}} \cdot d_\ell(I)$. By Lemma 13, nodes $E \cdot S \cdot d_{\overline{\ell}} \cdot d_\ell$ and $E \cdot \hat{S} \cdot \hat{d}_{\overline{\ell}} \cdot d_\ell$ have the same valence. This contradicts Lemma 18, since P_ℓ takes the first step in $\hat{S}$, while $P_{\overline{\ell}}$ takes the first step in S. □

We now describe an algorithm that solves Equality Negation for two processes Q_0 and Q_1, and uses one booster object fewer than $\mathbf{A}_{en}$. Intuitively, the algorithm works as follows: Each process Q_k simulates the actions of the corresponding process P_k in an execution of $\mathbf{A}_{en}$, and uses this execution to determine what value to decide. In this simulation, the processes are allowed to access all objects used by $\mathbf{A}_{en}$, except $\mathcal{O}$, each of which is initialised to the state it has in configuration $E(I)$.

The simulation of P_k by Q_k starts by pretending that P_k is in the state it has in configuration $E(I)$. 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 an ENROLL(i) operation, P_k pretends that it applied this operation to $\mathcal{O}$ and received as the response the initial value for which Q_k must solve Equality Negation. In other words, Q_k pretends that when P_k enrolls to $\mathcal{O}$ it receives as the "challenge" input with which to solve Equality Negation just the input value for which Q_k is required to solve Equality Negation! Q_k then updates the (simulated) state of P_k as if this step were executed and proceeds with the simulation of the next operation of P_k.

- If *op* is a REVEAL(i) operation, Q_k does *not* simulate any more steps of $\mathbf{A}_{en}$. Instead, it uses i (the parameter of REVEAL that P_k is "about" to execute) as its decision.

The algorithm is described more formally in Figure 10. A few remarks about conventions used in the pseudocode are in order. Recall, from Figure 6, that Apply(P_k, *op*, O) is the procedure by which P_k invokes operation *op* on object O; it returns the result of

Shared: every object used in algorithm $\mathbf{A}_{en}$ except $\mathcal{O}$, initialised to the state it has in $E(I)$

S : auxiliary variable : schedule of $\mathbf{A}_{en}$ applicable to $E(I)$, initially empty

For process Q_k, $k \in \{0, 1\}$

1 $\quad$ init := initial value of Q_k
2 $\quad$ state := state of P_k in $E(I)$
3 $\quad$ **while** Q_k has not decided **do**
4 $\quad\quad$ $\langle op, O\rangle$:= NextOp(P_k, state)
5 $\quad\quad$ **if** $O \neq \mathcal{O}$ **then**
6 $\quad\quad\quad$ $[\![\ r := \mathrm{Apply}(P_k, op, O);\ \ S := S \cdot (P_k, op, O, r)\ ]\!]$
7 $\quad\quad\quad$ state := NextState(P_k, state, r)
8 $\quad\quad$ **else** (* $O = \mathcal{O}$ *)
9 $\quad\quad\quad$ **if** op = ENROLL(i) **then**
10 $\quad\quad\quad\quad$ $S := S \cdot (P_k, \text{ENROLL}(i), \mathcal{O}, \mathit{init})$
11 $\quad\quad\quad\quad$ state := NextState(P_k, state, init)
12 $\quad\quad\quad$ **else** (* op is a REVEAL operation to $\mathcal{O}$ *)
13 $\quad\quad\quad\quad$ let i be such that op = REVEAL(i)
14 $\quad\quad\quad\quad$ **decide** i

Figure 10: Solving Equality Negation for two processes using $\mathbf{A}_{en}$ with one fewer booster object.

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 $\mathbf{A}_{en}$ (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 $\mathbf{A}_{en}$ when in state s is to apply op to object O.

- NextState(P_k, s, r): returns the state that P_k enters in $\mathbf{A}_{en}$ if it receives response r from the operation it invokes when in state s.

In the algorithm we also use an auxiliary variable S whose value, informally speaking, is the schedule of steps of $\mathbf{A}_{en}$ 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 (cf. Lemma 25). In one atomic step, besides accessing an ordinary (local or shared) variable, each process can also modify the auxiliary variable S. To emphasise this, we bracket with "$[\![\cdots]\!]$" the line that corresponds to an atomic step affecting both an ordinary variable and the auxiliary variable S (line 6).

Lemma 25 *If $\mathbf{A}_{en}$ is an Equality Negation algorithm for two processes, then the algorithm in Figure 10 also solves Equality Negation for two processes, using one fewer* booster *object.*

PROOF. It is obvious from Figure 10 that the algorithm uses one booster object less than $\mathbf{A}_{en}$. It remains to show that it solves Equality Negation. Consider any execution of the algorithm. Let S_i be the value of the auxiliary variable S when the schedule it contains consists of exactly i steps. A straightforward induction on i shows that for all $i \geq 0$ such that S_i is defined,

(a) S_i is a schedule of $\mathbf{A}_{en}$ that is applicable to $E(I)$ and contains no REVEAL step to $\mathcal{O}$, and

(b) 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* 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 P_k in configuration $E \cdot S_j(I)$ of $\mathbf{A}_{en}$, for any j, $i \leq j < i+$.

By (a), the algorithm in Figure 10 is wait-free and satisfies Termination. Otherwise, for some $k \in \{0,1\}$, the while loop of Q_k does not terminate, and hence S_i is defined for all integers $i \geq 0$. But then there is an infinite schedule S^* of $\mathbf{A}_{en}$ that is applicable to $E(I)$; this contradicts the fact that $\mathbf{A}_{en}$ is wait-free.

It remains to show that the algorithm in Figure 10 satisfies Negation. Consider an execution of the algorithm in which both processes decide, and let v_0 and v_1 be the initial values of Q_0 and Q_1, respectively. Let $state_k$ be the value of local variable *state* of process Q_k when Q_k decides, and i_k be the decision of Q_k. Let S^* be the value of S when Q_0 and Q_1 have both decided (i.e., S^* is the final value of the auxiliary variable S). By lines 12–14, the step that P_k executes next when in state $state_k$ is to apply REVEAL(i_k) to $\mathcal{O}$. From (b), it follows that the operation of P_k pending in $E \cdot S^*(I)$ is a REVEAL(i_k) operation to $\mathcal{O}$. By (a), $E \cdot S^*$ is a node in $\mathcal{F}$. By Lemma 23, each process has exactly one ENROLL step to $\mathcal{O}$ in S^*. By line 10, the response of P_k's ENROLL step to $\mathcal{O}$ in S^* is the initial value of Q_k, v_k, for each $k \in \{0,1\}$. By Lemma 24, $v_0 = v_1$ iff $i_0 \neq i_1$, as wanted. □

We now have all the pieces needed to prove the main result of this section.

Theorem 4 *Type* booster *has Consensus number one.*

PROOF. Suppose, to the contrary, that booster has Consensus number at least two. Hence, we can solve Consensus for two processes using only booster objects and registers. By Corollary 1, we can solve Equality Negation for two processes using only booster objects and registers. Let $\mathbf{A}_{en}$ be an Equality Negation algorithm for two processes that uses a minimal number of booster objects (and any number of registers). By Theorem 1, $\mathbf{A}_{en}$ uses at least one booster object. By Lemma 25, we can solve Equality Negation for two processes using one fewer booster object than that in $\mathbf{A}_{en}$. This contradicts the definition of $\mathbf{A}_{en}$. $\square$

9 Concluding remarks

We have shown that the Consensus hierarchy is not robust if nondeterministic object types are allowed. In contrast, if only deterministic types are allowed, the Consensus hierarchy is robust [1, 15]. Although most of the common types are deterministic, some useful types are most naturally specified as nondeterministic ones. For example, consider a priority queue where keys are not unique. If several elements are in the queue with the same maximum priority, it is natural to allow the priority queue to remove and return any one of these elements. It is, of course, always possible to turn a nondeterministic type into a deterministic one by restricting the allowable behaviour of the type. This, however, results in unnecessary restrictions for the implementor of the type. As pointed out in [11], by arbitrarily and artificially excluding legitimate behaviour we may be inadvertently ruling out efficient implementations of the type. Thus, the investigation of properties of nondeterministic types is of practical relevance.

In view of our result, it is natural to inquire whether there is some other hierarchy which is robust and places object types in levels according to their power in supporting implementations of other types. As we mentioned in Section 1.2, Jayanti proved that only the Consensus hierarchy, *or a coarsening of it*, can be such a hierarchy. It is therefore consistent with our result that, by collapsing levels one and two of the Consensus hierarchy into a single level, we might get a robust hierarchy that can be used as the basis of comparing the power of types. In Part II of this paper, we prove that this is not possible. More specifically, we give a generalisation of types negation and booster, parameterised by any positive integer n, so that $\mathsf{negation}_n$ and $\mathsf{booster}_n$ are at level one of the Consensus hierarchy, but, when combined, they can implement any type in a system of n processes. Combined with Jayanti's result, this shows that, in the context of nondeterministic types, there is no hierarchy that arranges types according to their ability to implement other types which is both robust and nontrivial (i.e., has at least two levels).

Acknowledgements

Our thinking on shared-memory distributed computing has been deeply influenced from discussions with Tushar Chandra, Prasad Jayanti and Sam Toueg. Faith Fich made many helpful comments on an earlier draft of this paper. Patricia O'Brien suggested the title.

References

[1] Elizabeth Borowsky, Eli Gafni, and Yehuda Afek. Consensus power makes (some) sense! In *Proceedings of the 13th ACM Symposium on Principles of Distributed Computing*, pages 363–372, August 1994.

[2] Tushar Chandra, Vassos Hadzilacos, Prasad Jayanti, and Sam Toueg. Wait-freedom vs. t-resiliency and the robustness of the wait-free hierarchies. In *Proceedings of the 13th ACM Symposium on Principles of Distributed Computing*, pages 334–343, August 1994.

[3] Edger Dijkstra. Solution of a problem in concurrent programming control. *Communications of the ACM*, 8:569, 1965.

[4] Danny Dolev, Cynthia Dwork, and Larry Stockmeyer. On the minimal synchronism needed for distributed consensus. *Journal of the ACM*, 34(1):77–97, January 1987.

[5] Michael Fischer, Nancy Lynch, and Michael Paterson. Impossibility of distributed consensus with one faulty process. *Journal of the Association for Computing Machinery*, 32(2):374–382, April 1985.

[6] Maurice Herlihy. Impossibility results for asynchronous PRAM. In *Proceedings of the 3th ACM Symposium on Parallel Algorithms and Architectures*, July 1991.

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

[8] Maurice Herlihy and Jeannette Wing. Linearizability: A correctness condition for concurrent objects. *ACM Transactions on Programming Language and System*, 12(3):463–492, July 1992.

[9] Prasad Jayanti. On the robustness of Herlihy's hierarchy. In *Proceedings of the 12th Annual ACM Symposium of Principles of Distributed Computing*, pages 145–157, 1993.

[10] Jon Kleinberg and Sendhil Mullainathan. Resource bounds and combinations of consensus objects. In *Proceedings of the 12th Annual ACM Symposium of Principles of Distributed Computing*, pages 133–143, 1993.

[11] Barbara Liskov and John Guttag. *Abstraction and specification in program development.* MIT Press, Cambridge, Massachusetts, 1986.

[12] Wai-Kau Lo. More on t-resilence vs. wait-freedom. In *Proceedings of the 13th ACM Symposium on Principles of Distributed Computing*, pages 110–119, August 1995.

[13] 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.

[14] Shlomo Moran and Lihu Rappoport. On the robustness of h_m^r. In *Proceedings of the 10th Workshop on Distributed Algorithms*, October 1996.

[15] Gary Peterson, Rida Bazzi, and Gil Neiger. A gap theorem for consensus types. In *Proceedings of the 13th ACM Symposium on Principles of Distributed Computing*, pages 344–353, August 1994.

[16] Eric Schenk. The consensus hierarchy is not robust. 1996.