Generalized irreducibility of consensus and the equivalence of t-resilient and wait-free implementations of consensus*

Tushar Chandra[†] Vassos Hadzilacos[‡] Prasad Jayanti[§] Sam Toueg[¶]

October 30, 2001

Abstract

We study the consensus problem, which requires multiple processes with different input values to agree on one of these values, in the context of asynchronous shared memory systems. Prior research focussed either on t-resilient solutions of this problem (which must be correct even if up to t processes crash) or on wait-free solutions (which must be correct despite the crash of any number of processes). In this paper, we show that these two forms of solvability are closely related. Specifically, for all $n > t \geq 2$ and all sets $\mathcal{S}$ of shared object types (that include simple read/write registers), there is a t-resilient solution to n-process consensus using objects of types in $\mathcal{S}$ if and only if there is a wait-free solution to $(t+1)$-process consensus using objects of types in $\mathcal{S}$.

Our proof of this equivalence uses another result derived in this paper, that is of independent interest. Roughly speaking, this result states that a wait-free solution to $(n-1)$-process consensus is never necessary in designing a wait-free solution to n-process consensus, regardless of the types of objects available. More precisely, for all $n \geq 2$ and all sets $\mathcal{S}$ of shared object types (that include simple read/write registers), if there is a wait-free solution to n-process consensus that uses a wait-free solution to $(n-1)$-process consensus and objects of types in $\mathcal{S}$ then there is a wait-free solution to n-process consensus that uses only objects of types in $\mathcal{S}$.

1 Introduction

We consider concurrent systems in which asynchronous processes communicate via typed shared objects. Informally, an object's type specifies: (i) the number of ports, which represents the maximum number of processes that may access the object *simultaneously*; (ii) the

*A preliminary version of this paper appeared in the Proceedings of the 13th Annual ACM Symposium on Principles of Distributed Computing, August 14-17, 1994.

†IBM T. J. Watson Research Center, `tushar@watson.ibm.com`

‡University of Toronto, `vassos@cs.toronto.edu`, partially supported by a grant from the Natural Sciences and Engineering Research Council of Canada.

§Dartmouth College, `prasad@cs.dartmouth.edu`, supported by NSF RIA grant CCR-9410421 and Dartmouth College Startup grant.

¶University of Toronto, `sam@cs.toronto.edu`, supported by NSF grant CCR-9102231.

set of states of the object; (iii) the set of operations that processes may apply to the object through its ports; and (iv) the behavior of the object, described by the effect of each operation on the object's state and the value the operation returns, assuming no other operation is accessing the object at that time. Every object is *linearizable* [HW90]: when operations are invoked concurrently at different ports, the object behaves *as if* each operation had occurred instantaneously, with no interference from other operations, at some point between the time it was invoked and the time it returned its response. An object that belongs to a type with n ports is called *n-ported.*

In such systems, some shared objects, such as registers and test&set objects, are supported in hardware, while other objects, such as queues and stacks, are implemented in software. Objects used in the implementation of another object (registers and test&set objects, in our example) are called *base* objects with respect to that implementation. We consider two forms of implementations, wait-free and t-resilient. An implementation is *wait-free* if every process can complete every operation on the implemented object in a finite number of its own steps, regardless of whether other processes are fast, slow, or have crashed [Lam77, Pet83, Her91]. Wait-free implementations provide an extreme degree of fault-tolerance. They assure that even if just one process survives, it will be able to complete its operations on the implemented object. In contrast, *t-resilient* implementations support a more modest degree of fault-tolerance [FLP85, DDS87, LAA87]. They guarantee that nonfaulty processes will complete their operations, as long as no more than t processes fail, where t is a specified parameter. It is immediate from the definitions that wait-freedom is equivalent to $(n-1)$-resilience, where n is the number of processes in the system.

This paper concerns t-resilient and wait-free implementations of a particular type of object, known as n-consensus. Informally, an n-consensus object allows each of n processes to access it by proposing a value; the object returns the same value to all accesses, where the value returned is the value proposed by some process. The following are two reasons why it is important to implement n-consensus objects:

- It is possible to design a wait-free implementation of an object of *any* type, shared by n processes, using only n-consensus objects and registers [Her91].[1]
- In an asynchronous system, since processes progress at independent and arbitrarily varying speeds, the view that a process holds of the global state of the computation does not necessarily coincide either with the reality or with the views of other processes. However, processes can reconcile their differences and arrive at a mutually acceptable common view if they have access to consensus objects.

A lot of research was aimed at determining the feasibility of implementing n-consensus

[1] A register is an object that allows processes to access it (only) through write operations, each of which stores a new value into the register, and read operations, which return the current value of the register. It is possible to implement k-ported registers, for any k, from just 2-ported registers. This is a consequence of the well-known fact that arbitrary multi-reader, multi-writer atomic registers (hence, k-ported registers, for any k) can be implemented from single-reader, single-writer (hence, 2-ported) atomic registers. (For more information on register constructions see, for example, [Lam86, Jay95].) Because of this, whenever we mention registers we do not bother to explicitly specify the number of ports: it is understood that any number of ports greater than one suffices.

objects from other types of objects. Some of it studied the feasibility of t-resilient implementations [FLP85, DDS87, LAA87], while others studied the feasibility of wait-free implementations [LAA87, Her91, CIL94]. To a large degree, the two questions, namely, the feasibility of t-resilient implementations and the feasibility of wait-free implementations, were treated separately, as if they had no particular relationship with each other (see Section 1.1, for an exception). The main contribution of this paper is to show that the two questions are closely related, as we explain below.

Consider the task of devising a t-resilient implementation of an n-consensus object. As n decreases, the fraction of nonfaulty processes on which the implementation can rely gets smaller, and the task therefore seems to become progressively more difficult. For example, a t-resilient implementation that works only when a majority of processes are nonfaulty cannot be used when n becomes smaller than $2t+1$. In the limit, when n becomes $t+1$, the task amounts to providing a *wait-free* implementation of the object. Thus, *prima faciae*, it seems that the ability of objects belonging to a set $\mathcal{S}$ of types to support a t-resilient implementation of an n-consensus object is greater than their ability to support a wait-free implementation of a $(t+1)$-consensus object. We show that this is not the case. Specifically, our result is:

> **Equivalence Theorem:** For all $n > t \geq 2$ and all sets $\mathcal{S}$ of types that include `register` ($\mathcal{S}$ may include nondeterministic types), there is a t-resilient implementation of an n-consensus object from objects of types in $\mathcal{S}$ if and only if there is a wait-free implementation of a $(t+1)$-consensus object from objects of types in $\mathcal{S}$.

One use of our theorem stems from the observation that proofs of impossibility of t-resilient implementations of consensus tend to be generally much harder than proofs of impossibility of wait-free implementations of consensus.[2] Our theorem allows one to conclude the impossibility of t-resilient implementations of n-consensus simply by establishing the impossibility of wait-free implementations of $(t+1)$-consensus. For instance, since there is no wait-free implementation of 3-consensus from queues and registers [Her91], our theorem implies that there is no 2-resilient implementation of n-consensus from queues and registers (for all $n \geq 3$). We present several other such applications of our theorem.

A key ingredient in our proof of the equivalence theorem discussed above is another result of this paper that is of interest in its own right. Roughly speaking, this result states that $(n-1)$-consensus objects are not necessary for a wait-free implementation of n-consensus, *no matter what other base objects may be available for the implementation.* Specifically, our result is:

> **Generalized Irreducibility Theorem for Consensus:** For all $n \geq 2$ and all sets $\mathcal{S}$ of types that include `register` ($\mathcal{S}$ may include nondeterministic types), if there is a wait-free implementation of an n-consensus object from $(n-1)$-consensus objects

[2] The difference in difficulty can be appreciated by comparing the proof that there is no wait-free implementation of a 3-consensus object from registers and 1-bit read-modify-write objects to the proof that there is no 2-resilient implementation of an n-consensus object from the same objects [LAA87]. The latter proof is much longer (three pages versus one page) and the arguments are more involved.

and objects of types in $\mathcal{S}$, then there is a wait-free implementation of an n-consensus object from objects of types in $\mathcal{S}$.

In other words, the $(n-1)$-consensus base objects can be eliminated from the implementation.

The above theorem is more general (and has a more complex proof) than the well-known irreducibility of consensus, stated as follows: for all $n \geq 2$, there is no wait-free implementation of an n-consensus object from $(n-1)$-consensus objects and registers [Her91, JT92]. To illustrate the difference between the two results, consider the following question: Is there a wait-free implementation of a 4-consensus object from 3-consensus objects, 4-ported queues, and registers? The irreducibility of consensus does not help answer this question, but our result does. Specifically, since there is no wait-free implementation of a 4-consensus object from 4-ported queues and registers [Her91], our result implies that the answer to the above question is no.

1.1 Related Work

As stated earlier, this paper has two main results, the generalized irreducibility theorem and the equivalence theorem. We discuss below prior research related to each of these results.

Our generalized irreducibility theorem is related to the following robustness question, posed in [Jay97]: Suppose that (a) there is no wait-free implementation of an n-consensus object from registers and objects of type T, and (b) there is no wait-free implementation of an n-consensus object from registers and objects of type T'. Does it then follow that there is no wait-free implementation of an n-consensus object from registers, objects of type T, and objects of type T'?

For the special case when one of T and T' is m-`consensus`, for any m, and the other type is arbitrary (it may even be nondeterministic), our generalized irreducibility theorem states that the answer to the robustness question is yes. For the case when both T and T' are deterministic, Borowsky, Gafni, and Afek [BGA94], and Peterson, Bazzi, and Neiger [PBN94] prove that the answer is yes. Neither result is strictly stronger than the other: our result restricts one of the types to be m-`consensus`, while theirs restricts both types to be deterministic. It is significant that our result applies to *all* types, not just deterministic ones, because nondeterministic types sometimes exhibit dramatically different properties. For instance, in sharp contrast to the results in [BGA94, PBN94] stated above, Lo and Hadzilacos prove that the answer to the robustness question is no if types may be nondeterministic [LH00a]. For different models, Chandra et al. [CHJT94], Moran and Rappoport [MR96], and Schenk [Sch96] also prove that the answer to the robustness question is no.

We now describe prior work related to our equivalence theorem. To our knowledge, Borowsky and Gafni are the first to relate t-resilient and wait-free implementations of tasks. To state their result we need to introduce a new object type, called n-ported k-set consensus. Informally, an object of this type allows each of n processes to access it by proposing a value. Each access returns some value that has been proposed to the object, subject to the requirement that the number of *different* values returned by all the accesses does not exceed k. Thus, an n-consensus object is the special case of this object where $k = 1$.

Borowsky and Gafni's result is that, for all $n > t$, if there is a t-resilient implementation of n-ported t-set consensus from registers then there is a wait-free implementation of $(t + 1)$-ported t-set consensus from registers [BG93a]. It was also shown (independently, in [BG93a, HS93, SZ93]) that there is no wait-free implementation of $(t + 1)$-ported t-set consensus from registers. In conjunction with this, Borowsky and Gafni's result implies that there is also no t-resilient implementation of n-ported t-set consensus from registers, for *all* $n > t$.

Our equivalence theorem differs from Borowsky and Gafni's result in fundamental ways. First, our result concerns t-resilient implementations of consensus, while theirs concerns t-resilient implementations of t-set consensus. Second, our result applies regardless of the types of objects used in the implementation, while their result requires the objects to be registers. The two results are independent in that neither implies the other.

The proofs of both results are based on simulation techniques whereby a small number of processes simulate a t-resilient algorithm originally designed for a larger number of processes, in a manner that preserves the resilience of the original algorithm. The two simulation techniques bear some superficial resemblance but they differ in substance, reflecting the differences between the results noted in the previous paragraph. The simulation we use to obtain the generalized irreducibility theorem (cf. proof of Lemma 5.1) applies only to consensus algorithms, while Borowsky and Gafni's simulation [BG93a, BGLR01] (as well as the simulation we use to obtain the equivalence theorem — cf. proof of Lemma 6.1) applies to a wider class of algorithms that is formally characterized in [BGLR01]. Also, the simulations we use to obtain the generalized irreducibility theorem and the equivalence theorem apply to algorithms that use arbitrary base objects, while Borowsky and Gafni's simulation applies only to algorithms that use registers. There is a variant of Borowsky and Gafni's simulation [BG93b, CR96] that applies to algorithms that use registers and k-set consensus objects. This variant, however, cannot be used to obtain our equivalence theorem because the algorithm that results from the simulation has lower resilience than the original, simulated algorithm.

Our equivalence theorem requires that $t \geq 2$. A result by Lo, strengthened further by Lo and Hadzilacos, proves that the "only if" direction of our theorem does not hold for $t = 1$: There exist nondeterministic [Lo95] and even deterministic [LH00b] object types which, together with registers, can provide a 1-resilient implementation of 3-consensus, but cannot provide a wait-free implementation of 2-consensus.

1.2 Organization

In Section 2, we describe the model. In Sections 3 and 4, we present the intermediate results needed to prove the generalized irreducibility theorem. This theorem and the equivalence theorem are then proved in Sections 5 and 6, respectively. We conclude with some remarks on the equivalence theorem in Section 7.

2 Model and Definitions

Our description of the model is somewhat informal. Herlihy [Her91] has shown how to formalize a similar model using I/O automata [LT88]. We use the following notation for sets of natural numbers: for any $i, j \in \mathbb{N}$, $[i..j] = \{k \in \mathbb{N} : i \leq k \leq j\}$.

2.1 Types

A *type* is a tuple $T = (n, OP, RES, Q, \delta)$, where n is a positive integer denoting the number of ports, OP is a set of operations, RES is a set of responses, Q is a set of states, and $\delta \subseteq Q \times OP \times [1..n] \times Q \times RES$ is the type's *sequential specification*. The number of ports corresponds to the maximum number of processes that can concurrently access an object of this type. The sequential specification describes the behavior of an object of this type in the absence of concurrency: If operation op is applied to port i of an object of type T when the object is in state q, the object can enter state q' and return response res if and only if $(q, op, i, q', res) \in \delta$. If, for each $(q, op, i) \in Q \times OP \times [1..n]$, the set $\{(q', res) : (q, op, i, q', res) \in \delta\}$ has at most one element, T is deterministic. In this paper we allow types to be nondeterministic, but we require that they exhibit *finite nondeterminism*: each of the above sets must be finite.

For example, the n-ported consensus type n-`consensus`[3] informally described in Section 1 can be formally defined as the tuple (n, OP, RES, Q, δ), where $OP = \{propose\ u : u \in \mathbb{N}\}$, $RES = \mathbb{N}$, $Q = \mathbb{N} \cup \{\perp\}$, and for each $i \in [1..n]$ and $u, v \in \mathbb{N}$, δ contains exactly the following tuples: $(\perp, propose\ u, i, u, u)$ and $(v, propose\ u, i, v, v)$. Clearly, this is a deterministic type.

2.2 Objects and Linearizability

An object O is an instance of a type initialized to a specified state. For each operation op and port i of its type, O provides an access procedure $\textsc{Apply}(op, i, O)$. This is the sole means by which operation op can be applied to port i of O. As explained above, the sequential specification of O's type describes the behavior of O when access procedures are applied sequentially. In general, however, access procedures *at different ports* of an object can be applied concurrently. Usually, the behavior of the object in this case is constrained by the assumption that the object is *linearizable* [HW90]. This means that if there are no concurrent accesses to the *same* port then the object behaves *as if* it had been initialized to the specified state and each access procedure occurred instantaneously at some point between its invocation and its response.[4] Occasionally it will be convenient to consider objects that are linearizable only if used in restricted ways. When discussing such objects we will explicitly state the conditions under which they are linearizable.

[3] Throughout the paper we use the `typewriter font` for type names.

[4] We emphasize that the linearizability assumption constrains the behavior of an object only if access procedures at the same port are applied sequentially. No assurances are given about what the object does if access procedures are applied concurrently to one of its ports. The object may fail to respond, or it may return arbitrary responses in that case.

In general, when we talk about an object we need to specify both its type and its initial state. However, in the case of consensus objects (i.e., objects of type n-`consensus`, for some n) we will assume, without explicitly saying so, that the initial state is $\perp$. This is because a consensus object initialized to any state other than $\perp$ is trivial: it remains in that state forever returning it to each invocation.

2.3 Implementation of Consensus

We now explain what is an *implementation* of a *target* object $\mathcal{O}$ of type T from a set A of *base* objects. The concept of implementation can be defined in a general way, for target objects of any type. In this paper, however, we are concerned exclusively with implementations of n-consensus objects. For simplicity and brevity we tailor our definitions specifically to such objects.

An *implementation of an n-consensus object $\mathcal{O}$ from a set A of objects* consists of a specification of access procedures $\textsc{Apply}(\mathit{propose}\ u, i, \mathcal{O})$, for each $u \in \mathbb{N}$ and $i \in [1..n]$. These access procedures can store and manipulate values in private variables using ordinary programming language constructs; in addition they can apply operations (only) to objects in A through the access procedures provided by those objects. To apply operation *op* to port i of a base object $O \in A$, an access procedure of the target object $\mathcal{O}$ invokes $\textsc{Apply}(op, i, O)$; when this operation finishes it returns a response. A *step* of the target object $\mathcal{O}$'s access procedure refers to the invocation of an operation at some port of a base object, the receipt of that operation's response, and (if relevant) the assignment of that response to a private variable of $\mathcal{O}$'s access procedure.

The implementation must satisfy certain safety properties (typically linearizability) and liveness properties (typically wait freedom or t-resilience). We will state the properties that must be satisfied by (concurrent) executions of $\mathcal{O}$'s access procedures. Before doing so, we need to clarify certain points about such executions.

First, we assume that at most one operation is applied to each port of $\mathcal{O}$. This assumption can be made without loss of generality because $\mathcal{O}$ is a consensus object: If multiple operations could be applied to a port, the response given by a port to the first *propose* operation applied to it can be stored in a private variable associated with the port and returned to any subsequent operation without involving accesses to any other shared objects (recall that operations to the same port must be applied sequentially).

Second, the concurrent executions we consider may contain one or more access procedures of $\mathcal{O}$ that have *not* run to completion. Taking such executions into account is necessary since we are interested in executions where some of the processes that invoke access procedures may crash. Therefore, given an execution, there are four mutually exclusive possibilities for an access procedure P (note that, in view of the previous paragraph, there can be at most one instance of P in an execution):

(a) P does not appear in the execution. This could be because in this execution no process had an interest in invoking P, or because the process that has an interest in invoking P has not gotten around to it yet (and perhaps never will because it has crashed).

(b) P is finite and incomplete in the execution, meaning that it has been invoked but has not returned a response. This could be because the process invoking P has not had a chance to run long enough or because it has crashed.

(c) P is complete (and, of course, finite) in the execution, meaning that P has returned a response.

(d) P is infinite (and, of course, incomplete). In this case the process invoking P does not crash (since P has infinitely many steps in the execution), yet the access procedure does not terminate. A typical reason for this behavior is that processes that invoked other procedures crashed at inopportune moments, making it impossible for P to terminate. In this case we will say that the nonterminating access procedure is *blocked* in this execution.

With these comments in mind, consider a concurrent execution $\mathcal{E}$ of a consensus object $\mathcal{O}$'s access procedures. First we will define the safety properties that should hold in $\mathcal{E}$. We say that $\mathcal{O}$

- *satisfies validity in* $\mathcal{E}$ if the value returned by each access procedure in $\mathcal{E}$ was proposed by some access procedure in $\mathcal{E}$.
- *satisfies agreement in* $\mathcal{E}$ if the values returned by all access procedures in $\mathcal{E}$ are the same.

Recall that an n-consensus object $\mathcal{O}$ is linearizable in $\mathcal{E}$ if the values returned by the access procedures in $\mathcal{E}$ are in accordance with the sequential specification of n-`consensus`, when $\mathcal{O}$ is initialized to state $\perp$ and each access procedure in $\mathcal{E}$ takes effect atomically at some point after it is invoked and before it completes. It is easy to see that $\mathcal{O}$ satisfies validity and agreement in $\mathcal{E}$ if and only if it is linearizable in $\mathcal{E}$. Thus, instead of proving that an implementation of n-consensus is linearizable, we will prove that it satisfies validity and agreement.

Next we discuss the liveness properties of the implementation. Intuitively, wait freedom requires that if a process that invokes an access procedure of the target object $\mathcal{O}$ does not crash then it will receive a response, no matter how many processes invoking access procedures at other ports of $\mathcal{O}$ crash. The property of t-resilience requires that if a process that invokes an access procedure of the target object $\mathcal{O}$ does not crash then it will receive a response, as long as at most t processes invoking access procedures at other ports of $\mathcal{O}$ crash. Actually, there are two slightly different formulations of t-resilience. The weaker formulation assumes that all of $\mathcal{O}$'s ports must be accessed in an execution. Thus, if an access procedure does not appear in $\mathcal{E}$, the process that was supposed to invoke it is considered to have crashed. The stronger formulation of t-resilience does not make this assumption; here, an access procedure may not appear in $\mathcal{E}$ just because no process had an interest in invoking it in this execution. The difference between these two formulations lies in what counts as one of the up to t crashes that the implementation is supposed to tolerate. With respect to the four aforementioned possibilities (a)–(d) for an access procedure in an execution, in the weaker formulation access procedures in case (a) and (b) count as crashes; while in the stronger formulation only access procedures in case (b) count as crashes.

It turns out that our equivalence theorem holds under both definitions of t-resilience. It is clear that an implementation that is t-resilient under the strong definition is also t-resilient under the weak one. Thus, one direction of the theorem (namely, that a t-resilient implementation of n-consensus implies a wait-free implementation of $(t+1)$-consensus) is stronger under the weak definition of t-resilience, while the converse is stronger under the strong definition of t-resilience. The former direction has more interesting implications and so we focus on the weak definition of t-resilience (which makes the more interesting direction stronger), rather than bloat the paper with too-similar definitions and results. We will return to this issue briefly in Section 7.

We now state the liveness properties that should hold in $\mathcal{E}$. We say that $\mathcal{O}$

- *is wait-free for port i in $\mathcal{E}$* if for every $u \in \mathbb{N}$, $\textsc{Apply}(\textit{propose}\ u, i, \mathcal{O})$ is finite in $\mathcal{E}$.
- *is wait-free in $\mathcal{E}$* if it is wait-free for each port in $\mathcal{E}$.
- *is t-resilient in $\mathcal{E}$* if the access procedure at some port is infinite only if the access procedures in more than t other ports do not appear or are finite and incomplete in $\mathcal{E}$. (The strong definition of t-resilience simply omits the words "do not appear or" from this definition.)

For each of the safety and liveness properties defined above, we omit the qualifier "in $\mathcal{E}$" if the property holds in every concurrent execution of the target object's access procedures in which the base objects are linearizable.

2.4 Binding schemes

The *binding scheme* of an implementation refers to the rules that govern how each access procedure of the implementation's target object can apply operations to ports of the base objects — specifically, the number of ports of a base object to which it can apply operations, and the length of time during which it is permitted to apply operations to these ports. Under the most permissive binding scheme, called *softwired* binding [BGA94], an access procedure can apply operations to any number of ports of a base object, and it "owns" the port only for the duration of each operation. In this binding scheme, different access procedures of the target object may apply (at different times) operations to the same port. Under the more restrictive *one-to-one static binding scheme*, for each access procedure P and each base object O there is at most one port of O to which P can apply operations, in all executions of the implementation; moreover no other access procedure can apply operations to that port of O. With one-to-one static binding, we can think of each port of a base object as being "owned" by an access procedure of the target object, namely the one that is allowed to apply operations to that port.

$O_1, O_2, \ldots, O_\ell$: n-consensus objects, initialized to $\perp$

$\underline{\text{APPLY}(\textit{propose } u, i, \mathcal{O})}$; $u \in \mathbb{N}, i \in [1..m]$

$est_i := u$
for $j := 1$ **to** ℓ **do**
  if $i \in S_j$ **then**
    $est_i := \text{APPLY}(\textit{propose } est_i, \text{pos}(i, S_j), O_j)$
return est_i

Figure 1: Implementation of m-consensus object $\mathcal{O}$ from n-consensus objects

3 Achieving One-to-One Static Binding With Base Consensus Objects

In general, the binding of a target object $\mathcal{O}$ with its base objects is not one-to-one static. The main result of this section is that, if $\mathcal{O}$ has some base consensus objects then it is possible to transform the implementation so that, in the new implementation, the binding of $\mathcal{O}$ with its base consensus objects is one-to-one static. We begin by describing an intermediate implementation needed to prove this result.

For any $m > n$, we describe how to implement an m-consensus object $\mathcal{O}$ from n-consensus objects. The binding of $\mathcal{O}$ with its base objects is one-to-one static but our implementation is only conditionally correct: It is always wait-free, and it satisfies validity and agreement *provided that* no more than n of the m access procedures of $\mathcal{O}$ take steps.

Consider the n-element subsets of $[1..m]$. Let $S_1, S_2, \ldots, S_\ell$ be a listing of these subsets, where $\ell = \binom{m}{n}$. For all $i \in S_j$, define $\text{pos}(i, S_j) = k$ if i is the kth smallest element in S_j.

Our implementation of an m-consensus object $\mathcal{O}$, described in Figure 1, employs ℓ base n-consensus objects, denoted $O_1, O_2, \ldots, O_\ell$. The access procedure $\text{APPLY}(\textit{propose } v, i, \mathcal{O})$ is implemented as follows. For brevity, let P denote this access procedure. P keeps a running estimate of the eventual return value in a local variable est_i. Initially, this estimate is v, the value that P wants to propose. P considers the base objects $O_1, \ldots, O_\ell$ in sequence and performs the following actions. For each base object O_j, P checks if $i \in S_j$. If $i \notin S_j$, P does not access O_j. Otherwise, it proposes its current estimate to O_j (at port $\text{pos}(i, S_j)$ of O_j) and regards the return value as its new estimate. After considering all base objects, P regards the estimate as the response of $\mathcal{O}$.

Lemma 3.1 *The implementation of m-consensus object $\mathcal{O}$ in Figure 1 has the following properties:*

1. *The binding of $\mathcal{O}$ (with all its base objects) is one-to-one static.*

2. *$\mathcal{O}$ is wait-free.*

3. *$\mathcal{O}$ satisfies validity and agreement in all executions in which at most n access procedures take steps.*

PROOF. Part 1 follows from the observation that, port i of $\mathcal{O}$ applies an operation to port p of base object O_j if and only if $i \in S_j$ and $\text{pos}(i, S_j) = p$. Part 2 follows from the fact that the implementation has no unbounded loops.

For Part 3, consider an execution $\mathcal{E}$ where $i_1, i_2, \ldots, i_n$ are all the ports of $\mathcal{O}$ at which access procedures are invoked. (We may assume, without loss of generality, that access procedures in exactly n ports are invoked: if there is an execution that violates validity or agreement and involves access procedures in n' ports, where $n' \leq n$, then there is also an execution that violates validity or agreement, respectively, and involves access procedures in exactly n ports.) Specifically, let $\text{APPLY}(\textit{propose } u_k, i_k, \mathcal{O})$ be the access procedure executed at port i_k; for brevity, let P_{i_k} denote this access procedure. We argue below that $\mathcal{O}$ satisfies validity and agreement in $\mathcal{E}$.

> $\mathcal{O}$ satisfies validity: Using the fact that the base objects $O_1, \ldots, O_\ell$ satisfy validity, it follows by an easy induction that $\mathcal{O}$ satisfies validity.
>
> $\mathcal{O}$ satisfies agreement: Consider the object O_j such that $S_j = \{i_1, i_2, \ldots, i_n\}$. For each $k \in [1..n]$, P_{i_k} accesses O_j in its jth iteration of the for-loop. Since O_j returns the same response u to every access procedure that applies a propose operation to it, it follows that all access procedures have the same estimate u at the end of j iterations of the for-loop. That is, for all $k \in [1..n]$, $est_{i_k} = u$ just after P_{i_k} completes the jth iteration of the for-loop. Since $O_{j+1}, \ldots, O_\ell$ satisfy validity, it follows from the implementation that the estimate of an access procedure never changes from the $(j+1)$th iteration onwards. Thus, for every $k \in [1..n]$, P_{i_k} returns u.

This completes the proof of the lemma. $\square$

Lemma 3.2 *Let $m > n$ and $\mathcal{S}$ be any set of types. Consider a wait-free implementation of an m-consensus object $\mathcal{O}$ from objects belonging to types in $\mathcal{S}$. Suppose that $\mathcal{O}$ has an n-consensus base object O. If the binding of $\mathcal{O}$ with O is not one-to-one static, it is possible to modify the implementation of $\mathcal{O}$, by replacing O with a bounded number of n-consensus objects, in such a way that the new implementation satisfies validity and agreement, is wait-free, and the binding of $\mathcal{O}$ with each of the newly introduced n-consensus base objects is one-to-one static.*

PROOF. Suppose that the binding of $\mathcal{O}$ with O is not one-to-one static. Consider the following modifications to the implementation of $\mathcal{O}$:

1. The base objects of the new implementation are the same as in the original implementation with one exception: The base object O is replaced with O', where O' is implemented as in Figure 1. (Thus, O' is an m-consensus object, but it is implemented entirely from n-consensus objects.)

2. The access procedures of the new implementation are the same as in the original implementation with one exception: Each time an access procedure $\text{APPLY}(\textit{propose } v, i, \mathcal{O})$ performs, in the original implementation, an operation (say, *propose* u) at some port j of O, the new implementation requires the access procedure to perform the same operation (namely, *propose* u) at port i of O'.

It is obvious from the above modifications that, in the new implementation, port i of O' is used only by the access procedures for port i of $\mathcal{O}$. This, together with the fact that the binding between O' and its base objects is one-to-one static (by Part 1 of Lemma 3.1), implies that the binding between $\mathcal{O}$ and the newly introduced n-consensus base objects (i.e., the base objects of O') is one-to-one static.

The fact that the new implementation satisfies validity and agreement follows from two facts: (i) O, the base object of the old implementation, has only n ports, and (ii) O', which replaces O, satisfies validity and agreement if it is accessed at no more than n of its m ports (by Part 3 of Lemma 3.1).

That the new implementation is wait-free follows again from two facts: (i) the old implementation is wait-free, and (ii) the implementation of O' is wait-free (by Part 2 of Lemma 3.1). This completes the proof of the lemma. $\square$

4 The Building Blocks

In this section we present three implementations of n-consensus objects from $(n-1)$-consensus objects. These implementations are only conditionally correct: each ensures wait freedom, validity and agreement only in executions that satisfy certain conditions. Yet they have certain nice properties that make them useful in the proof of the generalized irreducibility theorem, presented in the next section.

4.1 Nonconcurrent Implementation

Figure 2 shows an implementation of n-consensus object $\mathcal{O}$ from two $(n-1)$-consensus objects O and O' and a register *DEC*. This implementation is wait-free and, if the access procedures for ports $n-1$ and n are not executed concurrently, the implementation satisfies validity and agreement.

The implementation is informally described as follows. For $i \in [1..n]$, let P_i denote the access procedure $\text{APPLY}(\textit{propose } u_i, i, \mathcal{O})$, where $u_i \in \mathbb{N}$. $P_1, \ldots, P_{n-2}$ share $\mathcal{O}$'s base objects O and O' with P_{n-1} and P_n, respectively. For each $i \in [1..n-2]$, P_i proposes its value u_i to O, proposes the return value from O to O', writes the return value from O' in the (decision) register *DEC*, and returns it. Each of P_{n-1} and P_n first checks if the return value is already available in *DEC*. If not, it proposes its value to the appropriate base consensus object (namely, O for P_{n-1} and O' for P_n), writes the return value from the base consensus object in *DEC*, and returns it.

We now explain intuitively why this implementation works. Since the implementation

O, O': $(n-1)$-consensus objects, initialized to $\perp$
DEC: register, initialized to $\perp$

$\underline{\textsc{Apply}(\textit{propose } u, i, \mathcal{O})}$; $u \in \mathbb{N}, i \in [1..n-2]$

$v_i := \textsc{Apply}(\textit{propose } u, i, O)$
$DEC := \textsc{Apply}(\textit{propose } v_i, i, O')$
return DEC

$\underline{\textsc{Apply}(\textit{propose } u, n-1, \mathcal{O})}$; $u \in \mathbb{N}$

if $DEC = \perp$ **then**
  $DEC := \textsc{Apply}(\textit{propose } u, n-1, O)$
return DEC

$\underline{\textsc{Apply}(\textit{propose } u, n, \mathcal{O})}$; $u \in \mathbb{N}$

if $DEC = \perp$ **then**
  $DEC := \textsc{Apply}(\textit{propose } u, n-1, O')$
return DEC

Figure 2: Nonconcurrent implementation of n-consensus object $\mathcal{O}$ from $(n-1)$-consensus objects

needs to be correct only if the steps of P_{n-1} and P_n do not overlap, there are two cases: either P_{n-1} is before P_n or P_n is before P_{n-1}. In the former case, $P_1, \ldots, P_{n-2}$ and P_{n-1} agree on a return value using the object O, and P_n learns this value simply by reading DEC, where the value is made available by P_{n-1}. In the latter case, O' serves as the object that brings about agreement on the return value among $P_1, \ldots, P_{n-2}$ and P_n, and P_{n-1} learns this value by reading DEC.

Lemma 4.1 *The implementation, shown in Figure 2, of the n-consensus object $\mathcal{O}$ from $(n-1)$-consensus objects and a register has the following properties:*

1. *$\mathcal{O}$ is wait-free.*
2. *$\mathcal{O}$ satisfies validity in all executions, and agreement in all executions in which the access procedures at ports $n-1$ and n of $\mathcal{O}$ are not concurrent.*

PROOF. Part 1 is obvious. For Part 2, let $\mathcal{E}$ be any execution of the implementation; for each $i \in [1..n]$, let P_i be the access procedure at port i of $\mathcal{O}$ that appears in $\mathcal{E}$. (We may assume, without loss of generality, that access procedures at all ports appear in $\mathcal{E}$: If there is any execution that violates validity or agreement, then there is also one that violates validity or agreement, respectively, and involves the access procedures at all ports.)

$\underline{\mathcal{O} \text{ satisfies validity}}$: This follows easily by induction and the fact that the base objects O and O' satisfy validity.

$\mathcal{O}$ satisfies agreement, if P_{n-1} and P_n are not concurrent: Suppose P_{n-1} and P_n are not concurrent in $\mathcal{E}$. We observe that each of O and O' satisfies agreement. Let d and d' denote the (unique) return values from O and O', respectively, in $\mathcal{E}$.

If the first of P_{n-1} and P_n to read DEC finds that $DEC \neq \perp$, it is clear from the implementation that some P_i, $i \in [1..n-2]$, had previously written d into DEC. Since O and O' satisfy agreement, it is easy to see that, in this case, all access procedures return d and so $\mathcal{O}$ satisfies agreement. If the first of P_{n-1} and P_n to read DEC finds that $DEC = \perp$, we consider two cases, depending on which of the two access procedures was executed first (recall that P_{n-1} and P_n are not concurrent in $\mathcal{E}$).

If P_{n-1} was executed first, it is clear from the implementation that P_{n-1} writes d (the return value from O) in DEC and returns it. P_n, which begins after P_{n-1} finishes, reads d from DEC and return it. $P_1, \ldots, P_{n-2}$ propose d to O'. Since O' satisfies validity, and d is the only value proposed to it, it returns d to all, and all of $P_1, \ldots, P_{n-2}$ therefore return d. Thus, $\mathcal{O}$ satisfies agreement.

If P_n was executed first, it is clear from the implementation that $P_1, \ldots, P_{n-2}$ and P_n return d' (the return value from O'). P_n also writes d' in DEC. P_{n-1}, which begins after P_n finishes, reads d' from DEC and returns it. Thus, $\mathcal{O}$ satisfies agreement.

This completes the proof of the lemma. □

4.2 Group-Solo Implementation

Figure 3 shows an implementation of an n-consensus object $\mathcal{O}$ from two $(n-1)$-consensus objects O and O', and three registers GP, SP, and DEC. This implementation is wait-free for all ports except port n. It is also wait-free for port n unless both of the following hold: (i) an operation is invoked on a port other than n, and (ii) no such operation completes. (Note that since ports $1, \ldots, n-1$ are wait-free, an operation that has been invoked on these ports can fail to complete only because of a crash.)

We now informally describe how this implementation works. For $i \in [1..n]$, let P_i denote the access procedure at port i of $\mathcal{O}$. $P_1, \ldots, P_{n-1}$ act as one group, while P_n acts as a solo outsider. $P_1, \ldots, P_{n-1}$ reach consensus on their initial proposals by accessing O. Each P_i in the group regards the response of O as the group's proposal for consensus with P_n, and writes this value into register GP (see line 1). (GP and SP are acronyms for "Group's Proposal" and "Solo Proposal", respectively, and DEC stands for "Decision".) P_i then reads SP to check if the solo access procedure P_n has published its proposal yet. If SP is blank, P_i attempts to promote the group's proposal as the consensus value between P_n and the group. Otherwise, P_i attempts to promote the value in SP (which is the proposal of the solo access procedure P_n) as the consensus value. Lines 2, 3 and 4 in which P_i sets the local variable $vote_i$ to either GP or SP implement this strategy. It is possible, however, that some access procedures in the group find SP blank and consequently promote the group's proposal, while others find in SP a nonblank value that they promote. To reconcile such differences, access procedures in the group reach consensus on their votes by accessing O'. The response of O' is regarded as the final outcome of consensus between P_n and the group.

O, O': $(n-1)$-consensus objects, initialized to $\bot$
GP, SP, DEC: registers, initialized to $\bot$

	$\underline{\textsc{Apply}(\mathit{propose}\ u, i, \mathcal{O})}$; $u \in \mathbb{N}, i \in [1..n-1]$	$\underline{\textsc{Apply}(\mathit{propose}\ u, n, \mathcal{O})}$; $u \in \mathbb{N}$
1.	$GP := \textsc{Apply}(\mathit{propose}\ u, i, O)$	$SP := u$
2.	**if** $SP = \bot$ **then**	**if** $GP = \bot$ **then**
3.	$\quad vote_i := GP$	$\quad DEC := u$
4.	**else** $vote_i := SP$	**else** busy wait until $DEC \neq \bot$
5.	$DEC := \textsc{Apply}(\mathit{propose}\ vote_i, i, O')$	
6.	**return** DEC	**return** DEC

Figure 3: Group-Solo implementation of n-consensus object $\mathcal{O}$ from $(n-1)$-consensus objects

The solo access procedure P_n, on the other hand, begins by publishing its proposal in register SP. It then reads GP, the register where the group's proposal is published. If GP is blank, P_n concludes that it is ahead of all access procedures in the group and that access procedures in the group will all vote for its (P_n's) proposal. Thus, P_n regards its proposal as the outcome of its consensus with the group. On the other hand, if P_n finds GP nonblank, then P_n is uncertain of the views of the access procedures in the group (because some members of the group might promote P_n's proposal, while the others promote the group's proposal). P_n therefore blocks itself until the consensus value is published in the register DEC by (some P_i in) the group.

Lemma 4.2 *The implementation, shown in Figure 3, of the n-consensus object $\mathcal{O}$ from $(n-1)$-consensus objects and registers has the following properties:*

1. *$\mathcal{O}$ satisfies validity and agreement.*
2. *$\mathcal{O}$ is wait-free for all ports except port n.*
3. *$\mathcal{O}$ is wait-free for port n in every execution where either no operation is applied to other ports or an operation applied to some port returns a response.*

Proof. For Part 1, let $\mathcal{E}$ be any execution of the implementation; for each $i \in [1..n]$, let P_i denote the access procedure at port i of $\mathcal{O}$ that appears in $\mathcal{E}$. (As in the proof of Lemma 4.1 we may assume, without loss of generality, that access procedures at all ports appear in $\mathcal{E}$.)

$\underline{\mathcal{O}\text{ satisfies validity:}}$ This follows easily by induction and the fact that the base objects O and O' satisfy validity.

$\underline{\mathcal{O}\text{ satisfies agreement:}}$ There are two cases, based on whether the if-condition on line 2 of P_n evaluates to true or to false. Below we consider these cases in turn.

Suppose that the if-condition (in P_n's access procedure) evaluates to true. Then, two facts are obvious from the implementation: (i) P_n finished writing its proposal u in *SP* before any of $P_1, \ldots, P_{n-1}$ completed line 1 of its access procedure, and (ii) P_n writes u in *DEC* and returns it. Fact (i) implies that, for each $i \in [1..n-1]$, the if-condition on line 2 of P_i evaluates to false, thus causing P_i to set $vote_i$ to u, the value in *SP*. Thus, all of $P_1, \ldots, P_{n-1}$ propose u to O'; since O' satisfies validity, O' returns u to all. Thus, all of $P_1, \ldots, P_{n-1}$ return u. Since P_n also returns u, we have agreement.

Suppose that the if-condition in P_n evaluates to false. Then, two facts are obvious from the implementation: (i) P_n waits until it reads a non-$\perp$ value in *DEC* and returns it, and (ii) the only writes performed on *DEC* are by $P_1, \ldots, P_{n-1}$ on line 5, when they write the return value of O' in *DEC*. Since O' satisfies validity and agreement, it returns the same non-$\perp$ value, say d, to all of $P_1, \ldots, P_{n-1}$. Thus, each of $P_1, \ldots, P_{n-1}$ writes d in *DEC* and returns it. P_n eventually reads d in *DEC* and returns it. Hence we have agreement.

Parts 2 and 3 are obvious from the implementation. □

4.3 1-Blocking Array Implementation

We now use the Group-Solo implementation of n-consensus to implement, for any constant c, an array $\mathcal{O}[1..c]$ of n-consensus objects from $(n-1)$-consensus objects and registers. To access the ith port of the jth object in the array, where $i \in [1..n]$ and $j \in [1..c]$, the implementation provides the access procedure APPLY(*propose* $u, i, \mathcal{O}[j]$). The implementation guarantees some nice properties, but only in certain restricted executions. The implementation treats port n of the array elements differently from ports $1, \ldots, n-1$ in two respects: the restrictions it places on how the port may be used, and the wait-free properties it guarantees for the port.

Specifically, the implementation imposes the following restriction for all ports $i \in [1..n-1]$: port i of two different array elements must not be accessed concurrently. More precisely, the execution of access procedures at port i of any two array elements $\mathcal{O}[j]$ and $\mathcal{O}[k]$ must not be concurrent. Concurrent execution of access procedures at port n of distinct array elements is, however, permitted.

In executions that satisfy the above restriction, the implementation satisfies validity and agreement and, in addition, guarantees the following two properties: (i) All of $\mathcal{O}[1], \ldots, \mathcal{O}[c]$ are wait-free for ports $1, \ldots, n-1$, and (ii) All but one of $\mathcal{O}[1], \ldots, \mathcal{O}[c]$ are wait-free for port n. By Property (ii), if access procedures at port n of two different objects in the array are executing concurrently, they do not *both* block. This property will be crucial to the proof of the generalized irreducibility theorem of the next section.

Below we develop the ideas behind the implementation in two stages. In the first stage, we propose an obvious implementation and point out its drawbacks. We fix these drawbacks in the second stage.

Stage 1, Obvious Implementation: Let $GS_1, \ldots, GS_c$ be n-consensus objects implemented using the Group-Solo implementation of Figure 3. APPLY(*propose* $u, i, \mathcal{O}[j]$) simply makes a call to APPLY(*propose* u, i, GS_j).

The drawback is that this implementation fails to satisfy Property (ii), stated above. To see this, consider an execution of the access procedure APPLY(*propose* $v_1, i_1, \mathcal{O}[1]$), for some v_1 and $i_1 \in [1..n-1]$, up to the point where register *GP* of GS_1 has been written but before register *DEC* of GS_1 has been written (see lines 1 and 5 in Figure 3). Thus, port n of GS_1 is now blocked: an access procedure that invokes an operation on port n of GS_1 will have to wait until register *DEC* of GS_1 is written (see the busy-wait statement in Figure 3). Suppose that, at this point, the access procedure APPLY(*propose* $v_2, i_2, \mathcal{O}[2]$), for some v_2 and $i_2 \in [1..n-1]$ such that $i_2 \neq i_1$, is executed, again up to the point where register *GP* of GS_2 has been written but before register *DEC* of GS_2 has been written. Thus, port n of GS_2 is now blocked as well. If access procedures are now called at port n of $\mathcal{O}[1]$ and $\mathcal{O}[2]$, they will *both* have to wait indefinitely, violating Property (ii).

Stage 2, Refined Implementation: We observe that the failure of the above implementation to satisfy Property (ii) is due to the fact that it permits port n of more than one *GS* object to become blocked. We now describe a mechanism which ensures that, at all times, port n of at most one *GS* object can be blocked. The basic idea is as follows: When an access procedure P wants to apply *propose* u at port $i \in [1..n-1]$ of $\mathcal{O}[j]$, as in the obvious implementation P applies *propose* u at port i of GS_j, but it does so only after completing any propose operations that have already been initiated by other access procedures at ports $1, \ldots, n-1$ of $GS_1, \ldots, GS_c$. In this way, before P accesses GS_j to perform its own operation (thereby potentially causing port n of that object to become blocked) it ensures that port n of every other object is not blocked.

This idea is implemented as follows. We use $(n-1)$-consensus objects $O_1, \ldots, O_c$ (in addition to $GS_1, \ldots, GS_c$). When an access procedure P wants to apply an operation *propose* u at a port $i \in [1..n-1]$ of $\mathcal{O}[j]$, P seeks to "obtain permission" to apply *propose* u at port i of GS_j. To obtain this permission, P accesses $O_1, \ldots, O_c$, in that order, as described below. P proposes the tuple $\langle j, u\rangle$ to O_1.[5] Let $\langle k, v\rangle$ be O_1's response. There are two cases: $j \neq k$ or $j = k$. Below, we handle the two cases in turn.

If $j \neq k$, it means that some access procedure has already obtained permission (from O_1) to apply *propose* v to GS_k (at one of its first $n-1$ ports). This operation on GS_k may not have completed, and thus it is possible that port n of GS_k is blocked. P therefore helps complete that operation by applying *propose* v on GS_k. P then accesses O_2 for permission to apply *propose* u on GS_j. It does this by proposing $\langle j, u\rangle$ to O_2, and proceeds as above.

If $j = k$, it means that P has the permission to apply a propose operation to GS_j. At this point, P can propose either u or v (u is justified since it is P's proposal to $\mathcal{O}[j]$ and v is justified since some access procedure wants to propose it to $\mathcal{O}[j]$). In our implementation, P proposes v to GS_j, and regards the response of GS_j as the response of $\mathcal{O}[j]$ to its *propose* u operation.

Lemma 4.3 *Consider the implementation of array $\mathcal{O}[1..c]$ of n-consensus objects, shown in Figure 4. Let $\mathcal{E}$ be any execution of this implementation that satisfies the following*

[5] Strictly speaking it is natural numbers, and not pairs of natural numbers, that can be proposed to consensus objects. It is well-known, however, that pairs of natural number can be "coded" by natural numbers, so this is not a problem.

$O_1, \ldots, O_c$: $(n-1)$-consensus objects, initialized to $\perp$
$GS_1, \ldots, GS_c$: n-consensus objects, implemented using the Group-Solo implementation in Figure 3 from $(n-1)$-consensus objects and registers

$\underline{\text{APPLY}(\textit{propose } u, i, \mathcal{O}[j])}$; $u \in \mathbb{N}, j \in [1..c]$, $i \in [1..n-1]$

$\ell := 1$
repeat
    $\langle k, v\rangle := \text{APPLY}(\textit{propose } \langle j, u\rangle, i, O_\ell)$
    $\textit{res} := \text{APPLY}(\textit{propose } v, i, GS_k)$
    $\ell := \ell + 1$
until $j = k$
return *res*

$\underline{\text{APPLY}(\textit{propose } u, n, \mathcal{O}[j])}$; $u \in \mathbb{N}, j \in [1..c]$

$\textit{res} := \text{APPLY}(\textit{propose } u, n, GS_j)$
return *res*

Figure 4: Implementation of 1-blocking array $\mathcal{O}[1..c]$ of n-consensus objects from $(n-1)$-consensus objects

property:

(A) *For all ports $i \in [1..n-1]$ and all $j, j' \in [1..c]$ such that $j \neq j'$, the access procedures at port i of $\mathcal{O}[j]$ and port i of $\mathcal{O}[j']$ are not concurrent in $\mathcal{E}$.*

The following hold in $\mathcal{E}$:

1. *All of $\mathcal{O}[1], \ldots, \mathcal{O}[c]$ are wait-free for ports $1, \ldots, n-1$.*
2. *All but one of $\mathcal{O}[1], \ldots, \mathcal{O}[c]$ are wait-free for port n.*
3. *All of $\mathcal{O}[1], \ldots, \mathcal{O}[c]$ satisfy validity and agreement.*

PROOF. We prove the lemma through a series of claims. The first two claims state that the base objects are accessed as required to ensure they behave properly. As a result, these objects satisfy their safety and liveness properties.

Claim 4.3.1 *For each $i \in [1..n-1]$ and $\ell \in [1..c]$ there are no concurrent operations applied to port i of O_ℓ in $\mathcal{E}$.*

PROOF OF CLAIM 4.3.1. The claim follows immediately from the following two facts: (i) port i of O_ℓ is accessed only by access procedures invoked at port i of $\mathcal{O}[1], \ldots, \mathcal{O}[c]$, and (ii) by (A), access procedures at port i of different objects $\mathcal{O}[j]$ and $\mathcal{O}[j']$ are not concurrent. $\square$ Claim 4.3.1

Claim 4.3.2 *For each $i \in [1..n]$ and $k \in [1..c]$, there are no concurrent operations applied to port i of GS_k in $\mathcal{E}$.*

PROOF OF CLAIM 4.3.2. For $i \in [1..n-1]$ the claim is immediate from the following two facts: (i) port i of GS_k is accessed only by access procedures invoked at port i of $\mathcal{O}[1], \ldots, \mathcal{O}[c]$, and (ii) by (A), access procedures are not executed concurrently at port i of different objects $\mathcal{O}[j]$ and $\mathcal{O}[j']$. For $i = n$, the claim follows from the observation that port n of GS_k is accessed only by the access procedure at port n of $\mathcal{O}[k]$. $\square$ Claim 4.3.2

Notice that each proposal to (and hence each response of) O_ℓ is of the form $\langle j, u \rangle$. The next claim states that the first components of the responses returned by different objects among $O_1, \ldots, O_c$ are different.

Claim 4.3.3 *Let $\langle j, u \rangle$ and $\langle j', u' \rangle$ be the values returned by operations applied to objects O_ℓ and $O_{\ell'}$. If $\ell \neq \ell'$ then $j \neq j'$.*

PROOF OF CLAIM 4.3.3. Suppose $\ell \neq \ell'$. Without loss of generality, we can assume that $\ell' < \ell$. By Claim 4.3.1, O_ℓ satisfies validity and so some access procedure P proposed $\langle j, u \rangle$ to O_ℓ. By the implementation it is clear that P is an access procedure of the form APPLY(*propose* $u, *, \mathcal{O}[j]$).[6] An inspection of this access procedure shows that, before P proposed $\langle j, u \rangle$ to O_ℓ, it proposed $\langle j, u \rangle$ to $O_{\ell'}$ and received a response $\langle k, v \rangle$ where $k \neq j$. By Claim 4.3.1, $O_{\ell'}$ satisfies agreement, and so $\langle k, v \rangle = \langle j', u' \rangle$. Therefore, $j \neq j'$, as wanted. $\square$ Claim 4.3.3

Claim 4.3.4 *For each $i \in [1..n-1]$ and $j \in [1..c]$, $\mathcal{O}[j]$ is wait-free for port i in $\mathcal{E}$.*

PROOF OF CLAIM 4.3.4. For each $\ell \in [1..c]$, O_ℓ is wait-free for port i since, by Claim 4.3.1, it is accessed properly. By Part 2 of Lemma 4.2, GS_j is wait-free for port i. Thus, it remains to show that the repeat loop of access procedure APPLY(*propose* $u, i, \mathcal{O}[j]$) in Figure 4 terminates. If that loop does not terminate after ℓ iterations, then the propose operations applied to $O_1, \ldots, O_\ell$ return pairs whose first components are different from each other (by Claim 4.3.3) and from j. Recall that the first components of propose operations applied to $O_1, \ldots, O_c$ are integers in $[1..c]$, and that these objects satisfy validity since, by Claim 4.3.1, they are accessed properly. Therefore, the loop terminates after at most c iterations. $\square$ Claim 4.3.4

Claim 4.3.5 *All but one of $\mathcal{O}[1], \ldots, \mathcal{O}[c]$ are wait-free for port n in $\mathcal{E}$.*

PROOF OF CLAIM 4.3.5. Suppose, for contradiction, that there exist $k, k' \in [1..c]$ such that $k \neq k'$ and neither of $\mathcal{O}[k], \mathcal{O}[k']$ is wait-free for port n in $\mathcal{E}$. It follows that neither of $GS_k, GS_{k'}$ is wait-free for port n in $\mathcal{E}$. By Part 3 of Lemma 4.2, the following holds for both GS_k and $GS_{k'}$ in $\mathcal{E}$: An operation has been applied to some port other than n and no operation applied to any port returns a response.

[6] We use a $*$ to denote a quantity whose value is immaterial for the argument at hand.

Let P_k and $P_{k'}$ denote access procedures that applied, respectively, an operation to a port other than n of GS_k and $GS_{k'}$. From the implementation, it is clear that P_k obtained a response of $\langle k, * \rangle$ from some O_ℓ (otherwise P_k would not have applied an operation to GS_k). Similarly, $P_{k'}$ obtained a response of $\langle k', * \rangle$ from some $O_{\ell'}$. Since the responses of O_ℓ and $O_{\ell'}$ to P_k and $P_{k'}$, respectively, are different (recall that $k \neq k'$), and since each of O_ℓ and $O_{\ell'}$ satisfies agreement (recall that, by Claim 4.3.1, these objects are accessed properly), it follows that $\ell \neq \ell'$. Without loss of generality, assume $\ell' < \ell$. From the implementation it is clear that P_k accessed $O_{\ell'}$ before accessing $\mathcal{O}_\ell$. Since $O_{\ell'}$ satisfies agreement, its response to P_k was $\langle k', * \rangle$. It is again clear from the implementation that upon obtaining this response P_k invoked an operation $\text{APPLY}(\mathit{propose}\ *, *, GS_{k'})$ and received the corresponding response before proceeding to access $O_{\ell'+1}$. But this contradicts the fact that no operation applied to any port of $GS_{k'}$ returns a response. $\square$ Claim 4.3.5

Claim 4.3.6 *For each $j \in [1..c]$, $\mathcal{O}[j]$ satisfies validity and agreement in $\mathcal{E}$.*

Proof of Claim 4.3.6. We observe that the value returned in $\mathcal{E}$ by any of $\mathcal{O}[j]$'s access procedures (namely, $\text{APPLY}(\mathit{propose}\ *, *, \mathcal{O}[j])$) is a response received from GS_j. By Claim 4.3.2 and Lemma 4.2, GS_j satisfies agreement in $\mathcal{E}$. Therefore, $\mathcal{O}[j]$ also satisfies agreement in $\mathcal{E}$.

Consider any access procedure P of $\mathcal{O}[j]$ that returns v in $\mathcal{E}$. To prove that $\mathcal{O}[j]$ satisfies validity in $\mathcal{E}$ it suffices to show that some access procedure proposed v to $\mathcal{O}[j]$. Since P returns v, it is clear from the implementation that GS_j returned v to P. By Claim 4.3.2 and Lemma 4.2, GS_j satisfies validity in $\mathcal{E}$. Thus, v was proposed to GS_j by some access procedure P'. By the implementation, this implies that one of the following two cases applies:

(i) P' is the access procedure $\text{APPLY}(\mathit{propose}\ v, n, \mathcal{O}[j])$, and proposes v to GS_j.

(ii) P' is an access procedure $\text{APPLY}(\mathit{propose}\ *, i', \mathcal{O}[j'])$, for some $i' \in [1..n-1]$ and $j' \in [1..c]$, and proposes v to GS_j after receiving $\langle j, v \rangle$ from some object O_ℓ. This means that some access procedure P'' proposed $\langle j, v \rangle$ to O_ℓ, and so P'' is an access procedure $\text{APPLY}(\mathit{propose}\ v, *, \mathcal{O}[j])$.

In either case, some access procedure proposed v to $\mathcal{O}[j]$, as wanted. $\square$ Claim 4.3.6

The three parts of Lemma 4.3 are immediate from Claims 4.3.4, 4.3.5, and 4.3.6, respectively. $\square$

5 Generalized Irreducibility Theorem for Consensus

In this section, we prove the generalized irreducibility theorem for consensus, stated as follows. For all $n \geq 2$ and all sets $\mathcal{S}$ of types that include `register`, if there is a wait-free implementation of an n-consensus object from $(n-1)$-consensus objects and objects of types in $\mathcal{S}$, then there is a wait-free implementation of an n-consensus object just from objects of types in $\mathcal{S}$. In other words, the base $(n-1)$-consensus objects can be eliminated from the implementation. Thus, $(n-1)$-consensus objects are not necessary to implement a wait-free n-consensus object, regardless of the base objects that are available for such an implementation.

We obtain this result by repeated application of a lemma stating that if n-consensus objects are helpful in implementing $(n+1)$-consensus objects, then $(n-1)$-consensus objects are helpful in implementing n-consensus objects.

Lemma 5.1 *For all $n \geq 2$ and all sets $\mathcal{S}$ of types that include* `register`*, if there is a wait-free implementation of an $(n+1)$-consensus object from n-consensus objects and objects of types in $\mathcal{S}$, then there is a wait-free implementation of an n-consensus object from $(n-1)$-consensus objects and objects of types in $\mathcal{S}$.*

Proof. Consider a wait-free implementation of an $(n+1)$-consensus object $\mathcal{O}$ from n-consensus objects and objects of types in $\mathcal{S}$. By König's lemma, the number of base objects of $\mathcal{O}$ is finite [BNP94].[7]

Consider any n-consensus base object O of $\mathcal{O}$. By Lemma 3.2, we can assume that the binding of $\mathcal{O}$ with O is one-to-one static. Since $\mathcal{O}$ has $n+1$ ports and O has n ports, some port of $\mathcal{O}$, say p, does not use any port of O. Let $i_1, i_2, \ldots, i_n$ be the remaining ports of $\mathcal{O}$ (i.e., the elements of $[1..n+1] - \{p\}$) listed in ascending order. We may assume without loss of generality (by renaming ports of O, if necessary) that ports $i_1, i_2, \ldots, i_n$ of $\mathcal{O}$ are bound, respectively, to ports $1, 2, \ldots, n$ of O. We may therefore classify the base objects of $\mathcal{O}$ into four categories:

(a) $A_1, \ldots, A_a$ are the n-consensus base objects that are not accessed by port $n+1$ of $\mathcal{O}$. Thus, ports $n-1$ and n of $A_1, \ldots, A_a$ are bound, respectively, to ports $n-1$ and n of $\mathcal{O}$.

(b) $B_1, \ldots, B_b$ are the n-consensus base objects that are not accessed by port n of $\mathcal{O}$. Thus, ports $n-1$ and n of $B_1, \ldots, B_b$ are bound, respectively, to ports $n-1$ and $n+1$ of $\mathcal{O}$.

(c) $C_1, \ldots, C_c$ are the n-consensus base objects that are not accessed by one of the first $n-1$ ports of $\mathcal{O}$. Thus, ports $n-1$ and n of $C_1, \ldots, C_c$ are bound, respectively, to ports n and $n+1$ of $\mathcal{O}$.

(d) $D_1, \ldots, D_d$ are the remaining base objects of $\mathcal{O}$ (these belong to the types in $\mathcal{S}$).

Modify the implementation of $\mathcal{O}$ as follows:

- Replace the base objects $A_1, \ldots, A_a$ and $B_1, \ldots, B_b$ with a 1-blocking array $O[1..a+b]$, implemented as in Figure 4. Objects $O[1..a]$ of the array are used in the place of $A_1, \ldots, A_a$ and $O[a+1..a+b]$ are used in the place of $B_1, \ldots, B_b$.

[7] It is here that we make use of the assumption that types exhibit finite nondeterminism.

- Replace the base objects $C_1, \ldots, C_c$ with $NC_1, \ldots, NC_c$, where each NC_i is implemented using the nonconcurrent implementation in Figure 2.
- The access procedures of the new implementation are the same as in the original implementation with the following exception: Each time any access procedure APPLY(*propose* $*, *, \mathcal{O}$) applies, in the original implementation, a *propose* u operation to port i of A_j, B_j, or C_j, the new implementation requires the access procedure instead to apply *propose* u to port i of $O[j]$, $O[a+j]$, or NC_j, respectively.

Claim 5.1.1 *The $(n+1)$-consensus object $\mathcal{O}$ in the new implementation described above has the following properties:*

1. *All base consensus objects of $\mathcal{O}$ are $(n-1)$-consensus objects and all other base objects of $\mathcal{O}$ belong to types in $\mathcal{S}$.*
2. *$\mathcal{O}$ satisfies validity and agreement in all executions in which operations are not executed concurrently at ports $n-1$ and n of NC_i, for all $i \in [1..c]$.*
3. *$\mathcal{O}$ is wait-free for ports $1, \ldots, n-1$.*
4. *$\mathcal{O}$ is wait-free for one of ports n and $n+1$.*

PROOF OF CLAIM 5.1.1. Part 1 follows from the fact that $O[1..a+b]$ and $NC_1, \ldots, NC_c$ are implemented from $(n-1)$-consensus objects and registers, the latter being in $\mathcal{S}$ by assumption.

To prove Part 2, consider any execution $\mathcal{E}$ of the new implementation of $\mathcal{O}$ in which operations are not executed concurrently at ports $n-1$ and n of any NC_i, $i \in [1..c]$. The following three facts imply that $\mathcal{O}$ satisfies validity and agreement in $\mathcal{E}$:

- The original implementation of $\mathcal{O}$ satisfies validity and agreement in all executions.
- By Part 3 of Lemma 4.3, $O[1], \ldots, O[a+b]$ (which have replaced $A_1, \ldots, A_a$ and $B_1, \ldots, B_b$ of the original implementation) satisfy validity and agreement in all executions that satisfy the following proviso: there are no concurrent invocations at the *same* port $p \in [1..n-1]$ of two *distinct* objects $O[j]$ and $O[k]$, where $j, k \in [1..a+b]$. This proviso is satisfied in $\mathcal{E}$ because (i) the first $n-1$ ports of each object $O[1], \ldots, O[a+b]$ are bound to the first $n-1$ ports of $\mathcal{O}$, respectively; and (ii) in the original implementation of $\mathcal{O}$, no access procedure invokes concurrent operations on distinct base objects — and, in particular, on the n-consensus base objects $A_1, \ldots, A_a$ and $B_1, \ldots, B_b$ that were replaced by $O[1], \ldots, O[a+b]$.
- By Part 2 of Lemma 4.1, $NC_1, \ldots, NC_c$ (which have replaced $C_1, \ldots, C_c$ of the original implementation) satisfy validity and agreement in $\mathcal{E}$.

Part 3 follows from the following facts: (i) $NC_1, \ldots, NC_c$ are wait-free (by Part 1 of Lemma 4.1), and (ii) $O[1], \ldots, O[a+b]$ are wait-free for ports $1, \ldots, n-1$ (by Part 1 of Lemma 4.3), and (iii) none of the first $n-1$ ports of $\mathcal{O}$ use port n of any of $O[1], \ldots, O[a+b]$.

Part 4 follows from the following facts: (i) all but one of $O[1], \ldots, O[a+b]$ are wait-free for port n (by Part 2 of Lemma 4.3), and (ii) port n of each of $O[1], \ldots, O[a]$ is used only by port n of $\mathcal{O}$, and port n of each of $O[a], \ldots, O[a+b]$ is used only by port $n+1$ of $\mathcal{O}$ (thus, port n of each $O[i]$ is used either by port n of $\mathcal{O}$ or by port $n+1$ of $\mathcal{O}$, but not by both). $\square$ Claim 5.1.1

Next we describe how to transform the new implementation of $\mathcal{O}$, the $(n+1)$-consensus object satisfying the properties in Claim 5.1.1, into an implementation of an n-consensus object $\mathcal{O}'$. Informally, the access procedure for each of the first $n-1$ ports of $\mathcal{O}'$ simply calls the access procedure at the corresponding port of $\mathcal{O}$ and returns that procedure's response. For port n, the access procedure of $\mathcal{O}'$ executes the access procedures of *both* port n and $n+1$ of $\mathcal{O}$, alternating between the two in such a manner that one of them is guaranteed to terminate and return a value; that value then becomes the response of the access procedure of port n of $\mathcal{O}'$.

To explain more precisely how $\mathcal{O}'$ works, we need to make some observations and introduce some terminology. Recall (see Figure 4) that to propose a value to port n of an object $O[j]$ in a 1-blocking array, we simply propose the same value to port n of an object GS_j in the group-solo implementation. If, in some execution, the latter enters the busy-wait statement (see Figure 3), we say that object $O[j]$ is *blocked.*

Here now, is how the access procedure $\text{APPLY}(\textit{propose}\ u, i, \mathcal{O}')$ works:

- For all $i \in [1..n-1]$, $\text{APPLY}(\textit{propose}\ u, i, \mathcal{O}')$ executes $\text{APPLY}(\textit{propose}\ u, i, \mathcal{O})$.
- $\text{APPLY}(\textit{propose}\ u, n, \mathcal{O}')$, the access procedure at port n of $\mathcal{O}'$, interleaves the execution of $\text{APPLY}(\textit{propose}\ u, n, \mathcal{O})$ and $\text{APPLY}(\textit{propose}\ u, n+1, \mathcal{O})$, using the rules below. For convenience, we let PROC_n and PROC_{n+1} denote $\text{APPLY}(\textit{propose}\ u, n, \mathcal{O})$ and $\text{APPLY}(\textit{propose}\ u, n+1, \mathcal{O})$, respectively. Note that, by construction of $\mathcal{O}$, PROC_n applies operations to port n of $O[1], \ldots, O[a]$ (as well as to port $n-1$ of $NC_1, \ldots, NC_c$, and possibly to ports of $D_1, \ldots, D_d$). Similarly, PROC_{n+1} applies operations to port n of $O[a+1], \ldots, O[a+b]$ (as well as to port n of $NC_1, \ldots, NC_c$, and possibly to ports of $D_1, \ldots, D_d$).
  (a) Begin by executing PROC_n.
  (b) Suspend PROC_n and resume PROC_{n+1} if and only if PROC_n accesses an object $O[j]$ that is blocked (for some $j \in [1..a]$). Similarly, suspend PROC_{n+1} and resume PROC_n if and only if PROC_{n+1} accesses an object $O[a+j]$ that is blocked (for some $j \in [1..b]$).
  (c) As soon as either PROC_n or PROC_{n+1} terminates and returns some value v, terminate $\text{APPLY}(\textit{propose}\ u, n, \mathcal{O}')$ and return v.

Claim 5.1.2 *The n-consensus object $\mathcal{O}'$, in the implementation described above, has the following properties:*

1. *All base consensus objects of $\mathcal{O}'$ are $(n-1)$-consensus objects, and all other base objects belong to types in $\mathcal{S}$.*

2. $\mathcal{O}'$ *satisfies validity and agreement.*

3. $\mathcal{O}'$ *is wait-free.*

PROOF OF CLAIM 5.1.2. Part 1 follows directly from Part 1 of Claim 5.1.1.

Each access procedure of $\mathcal{O}'$ merely returns the value of a corresponding access procedure of $\mathcal{O}$. By Part 2 of Claim 5.1.1, $\mathcal{O}$ satisfies validity and agreement as long as there are no concurrent operations at ports $n-1$ and n of each NC_i, for $i \in [1..c]$. Thus, to prove Part 2 of the present claim, it suffices to prove that the access procedures of $\mathcal{O}'$ do not apply concurrent operations to ports $n-1$ and n of NC_i, for every $i \in [1..c]$. To see why this is the case recall that ports $n-1$ and n of each NC_i are accessed only by PROC_n and PROC_{n+1}, respectively. By construction of $\mathcal{O}'$, only the access procedure of port n of $\mathcal{O}'$ executes PROC_n and PROC_{n+1}. Furthermore, while the access procedure of port n of $\mathcal{O}'$ is executing one of PROC_n or PROC_{n+1}, it has suspended execution of the other. Therefore, no concurrent operations are executed at ports $n-1$ and n of NC_i, as wanted.

Part 3 of Claim 5.1.1 implies that $\mathcal{O}'$ is wait-free for ports $1, \dots, n-1$, and Part 4 of Claim 5.1.1 implies that $\mathcal{O}'$ is wait-free for port n. Thus, $\mathcal{O}'$ is wait-free for all its n ports. $\square_{\text{Claim 5.1.2}}$

This completes the proof of Lemma 5.1. $\square$

Theorem 5.2 (Generalized Irreducibility Theorem for Consensus) *For all $n \geq 2$ and all sets $\mathcal{S}$ of types that include* `register`, *if there is a wait-free implementation of an n-consensus object from $(n-1)$-consensus objects and objects of types in $\mathcal{S}$, then there is a wait-free implementation of an n-consensus object from objects of types in $\mathcal{S}$.*

PROOF. Suppose there is a wait-free implementation of an n-consensus object from $(n-1)$-consensus objects and objects of types in $\mathcal{S}$. By repeated application of Lemma 5.1 we have that for all $k \in [2..n]$, there is a wait-free implementation of a k-consensus object from $(k-1)$-consensus objects and objects of types in $\mathcal{S}$. Composing all these implementations, we obtain a wait-free implementation of an n-consensus object from 1-consensus objects and objects of types in $\mathcal{S}$. Since 1-consensus objects have a trivial implementation,[8] we have a wait-free implementation of an n-consensus object from objects of types in $\mathcal{S}$. $\square$

6 Equivalence of t-Resilient and Wait-Free Implementations of Consensus

In this section, we apply the generalized irreducibility theorem to prove the following equivalence between t-resilient and wait-free implementations of consensus: For all $n > t \geq 2$ and all sets $\mathcal{S}$ of types that include `register`, a t-resilient implementation of an n-consensus

[8] A *propose* u operation on a 1-consensus object simply returns u.

$\mathcal{O}$: $(t+1)$-consensus object, initialized to $\perp$
R: register, initialized to $\perp$

$\underline{\text{APPLY}(\textit{propose } u, i, \mathcal{O}')}$; $u \in \mathbb{N}$, $i \in [1..t+1]$	$\underline{\text{APPLY}(\textit{propose } u, i, \mathcal{O}')}$; $u \in \mathbb{N}$, $i \in [t+2..n]$
$R :=$ APPLY(*propose* $u, i, \mathcal{O}$)	**repeat until** $R \neq \perp$
return R	**return** R

Figure 5: t-resilient implementation of n-consensus object $\mathcal{O}'$ from a wait-free $(t+1)$-consensus object

object from objects of types in $\mathcal{S}$ exists if and only if a wait-free implementation of a $(t+1)$-consensus object from objects of types in $\mathcal{S}$ exists. The "if" direction is easy and is proved in the following lemma.

Lemma 6.1 *For all $n > t \geq 1$ and all sets $\mathcal{S}$ of types that include* `register`, *if there is a wait-free implementation of a $(t+1)$-consensus object from objects of types in $\mathcal{S}$, then there is a t-resilient implementation of an n-consensus object from objects of types in $\mathcal{S}$.*

PROOF. Let $\mathcal{O}$ be a wait-free implementation of a $(t+1)$-consensus object. Figure 5 describes how to implement a t-resilient n-consensus object $\mathcal{O}'$ from $\mathcal{O}$ and a register R. Informally, the access procedures for ports $1, \ldots, t+1$ of $\mathcal{O}'$ access $\mathcal{O}$ and make the return value available in register R. The access procedures for the remaining ports of $\mathcal{O}'$ simply wait until a return value is available in R. Since $\mathcal{O}$ satisfies validity and agreement, it follows that $\mathcal{O}'$ also satisfies validity and agreement. Since the implementation of $\mathcal{O}$ is wait-free, if at least $n-t$ of the n access procedures of $\mathcal{O}'$ keep taking steps, then the access procedure of at least one port $i \in [1..n]$ of $\mathcal{O}'$ completes its propose operation at port i of $\mathcal{O}$ and writes $\mathcal{O}$'s response in R. Thus, access procedures of $\mathcal{O}'$ that repeatedly take steps eventually terminate and return a value. Hence $\mathcal{O}'$ is t-resilient. $\square$

We now prove the converse of the above lemma, for $t \geq 2$.

Lemma 6.2 *For all $n > t \geq 2$ and all sets $\mathcal{S}$ of types that include* `register`, *if there is a t-resilient implementation of an n-consensus object from objects of types in $\mathcal{S}$, then there is a wait-free implementation of a $(t+1)$-consensus object from objects of types in $\mathcal{S}$.*

PROOF. Let $\mathcal{O}$ be a t-resilient implementation of an n-consensus object from objects of types in $\mathcal{S}$. Using $\mathcal{O}$ (and a few other base objects) we implement a wait-free $(t+1)$-consensus object $\mathcal{O}'$. Roughly speaking, the access procedures of $\mathcal{O}'$ coordinate with each other to simulate an execution of $\mathcal{O}$. The simulation is done in such a way that each access procedure of $\mathcal{O}'$ that crashes can prevent *at most one* access procedure of $\mathcal{O}$ in the simulated execution from making progress. Thus, if access procedures in at most t ports of $\mathcal{O}'$ crash then access procedures of at most t ports of $\mathcal{O}$ will stop making progress in the simulated execution. Since $\mathcal{O}$ is t-resilient, the access procedures in the remaining $n-t$ ports of $\mathcal{O}$ eventually terminate and return the same value. This value is adopted as the return value of $\mathcal{O}'$.

We implement the above idea as follows:

- We employ registers $U_1, \ldots, U_n$ and $R_1, \ldots, R_n$. For each $i \in [1..n]$, U_i stores the value proposed at port i of $\mathcal{O}$, and R_i stores the current state of that port's access procedure. (The state of an access procedure consists of the values of all its private variables and of its "program counter".)

- We employ $(t+1)$-ported test&set objects $TS_1, \ldots, TS_n$. (The test&set object type has two states—*win* and *lose*—and supports two operations: *test&set* and *reset*. The *test&set* operation returns the current state of the object, and sets the state to *lose*. The *reset* sets the state to *win*, and returns an acknowledgement as a response.)

  We use TS_j to ensure that at any time at most one of $\mathcal{O}'$'s access procedures simulates steps of the access procedure at port j of $\mathcal{O}$.

- For brevity, let P_i denote the access procedure APPLY(*propose* $u, i, \mathcal{O}'$), for each $i \in [1..t+1]$. P_i considers the n ports of $\mathcal{O}$ in round-robin fashion. When P_i considers port j of $\mathcal{O}$, it applies a *test&set* operation to TS_j. If TS_j returns *lose*, P_i moves on to consider the next port of $\mathcal{O}$, $\big((j+1) \mod n\big)+1$. If TS_j returns *win*, P_i advances the simulation of the access procedure at port j of $\mathcal{O}$ by performing the following actions:

  (1) P_i reads R_j to determine the current state of that procedure.

  (2) If no step of port j has been simulated before, P_i writes its proposal u into U_j, and sets R_j to the initial state of APPLY(*propose* $U_j, j, \mathcal{O}$).

  (3) P_i performs a single step of APPLY(*propose* $U_j, j, \mathcal{O}$) and writes the resulting state of that procedure in R_j.

  (4) P_i performs a reset operation on TS_j (so that some $P_{i'}$, $i' \in [1..t+1]$, may execute the next step of the access procedure at $\mathcal{O}$'s port j).

  (5) If the step that P_i simulated caused the access procedure of port j of $\mathcal{O}$ to terminate and return v, P_i writes v in a register DEC and terminates; otherwise, P_i moves on to consider the next port of $\mathcal{O}$.

The implementation of $\mathcal{O}'$, described informally above, is shown in Figure 6. The following claim states the desired properties of $\mathcal{O}'$.

Claim 6.2.1 *The $(t+1)$-consensus object $\mathcal{O}'$, shown in Figure 6, satisfies validity and agreement, and is wait-free.*

PROOF OF CLAIM 6.2.1. Let $\mathcal{E}'$ be an arbitrary concurrent execution of $\mathcal{O}'$'s access procedures, and let $\mathcal{E}$ be the subexecution of $\mathcal{E}'$ consisting of the operations applied to $\mathcal{O}$ (see line 8 in Figure 6). Thus $\mathcal{E}$ is a concurrent execution of $\mathcal{O}$'s access procedures. To

$\mathcal{O}$: t-resilient implementation of n-consensus object, initialized to $\perp$
$TS_1, \ldots, TS_n$: $(t+1)$-ported test&set objects, initialized to *win*
$R_1, \ldots, R_n$: registers, initialized to $\perp$
$U_1, \ldots, U_n$: registers, initialized arbitrarily
DEC: register, initialized to $\perp$

<u>$\text{APPLY}(\textit{propose } u, i, \mathcal{O}')$</u>; $u \in \mathbb{N}, i \in [1..t+1]$

1. $j := 1$
2. **repeat**
3.   **if** $\text{APPLY}(\textit{test\&set}, i, TS_j) = \textit{win}$ **then**
4.   read R_j to determine the state of PROC_j
5.   **if** $R_j = \perp$ **then**
6. $U_j := u$
7. $R_j :=$ initial state of $\text{APPLY}(\textit{propose } U_j, j, \mathcal{O})$
8.   execute one step of $\text{APPLY}(\textit{propose } U_j, j, \mathcal{O})$
9.   write in R_j the new state of $\text{APPLY}(\textit{propose } U_j, j, \mathcal{O})$
10. $\text{APPLY}(\textit{reset}, i, TS_j)$
11.   **if** $\text{APPLY}(\textit{propose } U_j, j, \mathcal{O})$ terminated and returned v **then** $\textit{DEC} := v$
12.   **if** $\textit{DEC} \neq \perp$ **then return** *DEC*
13. $j := \big((j+1) \mod n\big) + 1$
14. **forever**

Figure 6: Wait-free implementation of a $(t+1)$-consensus object $\mathcal{O}'$ from t-resilient implementation of n-consensus object $\mathcal{O}$

prove the claim it suffices to show that $\mathcal{O}'$ satisfies validity and agreement, and is wait-free in $\mathcal{E}'$.

It is clear from the implementation that the values returned by access procedures of $\mathcal{O}'$ in $\mathcal{E}'$ are values returned by access procedures of $\mathcal{O}$ in $\mathcal{E}$. In addition, any value proposed by an access procedure of $\mathcal{O}$ in $\mathcal{E}$ is a value proposed by some access procedure of $\mathcal{O}'$ in $\mathcal{E}'$ (see line 6). From these two observations, and the fact that $\mathcal{O}$ satisfies validity and agreement in $\mathcal{E}$, it follows that $\mathcal{O}'$ satisfies validity and agreement in $\mathcal{E}'$.

It remains to show that $\mathcal{O}'$ is wait-free in $\mathcal{E}'$. Suppose, for contradiction, that there is an access procedure, say P, of $\mathcal{O}'$ that is infinite in $\mathcal{E}'$. For $i \in [1..t+1]$ let P_i be the access procedure at port i of $\mathcal{O}'$ in $\mathcal{E}'$, and for $j \in [1..n]$ let Q_j be the access procedure at port j of $\mathcal{O}$ in $\mathcal{E}$.

For each $j \in [1..n]$, if Q_j is finite and incomplete in $\mathcal{E}$ then there is some $i \in [1..t+1]$ such that P_i applies only finitely many operations to TS_j in $\mathcal{E}'$, the last of which is a *test&set* that returns *win*. (Otherwise, since P applies infinitely many *test&set* operations to TS_j in $\mathcal{E}'$, there would be infinitely many *test&set* operations to TS_j that return *win* in

$\mathcal{E}'$ and Q_j would be infinite in $\mathcal{E}$.) Thus, at most t of $Q_1, \ldots, Q_n$ are finite and incomplete in $\mathcal{E}$. (Otherwise, more than t of $P_1, \ldots, P_{t+1}$ — and therefore all of them — are finite in $\mathcal{E}'$, contradicting that P is infinite in $\mathcal{E}'$.) Furthermore, each access procedure of $\mathcal{O}$ that is not finite and incomplete in $\mathcal{E}$ is infinite in $\mathcal{E}$. (Such an access procedure Q_j must appear in $\mathcal{E}$ because P considers Q_j infinitely often, and cannot be finite and complete in $\mathcal{E}$ because then the P_i that executed the last step of Q_j would write the response of Q_j in DEC and P would not be infinite in $\mathcal{E}'$.) Therefore, at most t of $Q_1, \ldots, Q_n$ are finite and incomplete and the remaining $n - t$ are infinite in $\mathcal{E}$. Since $n > t$, this contradicts that $\mathcal{O}$ is t-resilient. Thus, $\mathcal{O}'$ is wait-free in $\mathcal{E}'$. $\square$ Claim 6.2.1

Afek, Weisberger, and Weisman showed that there is a wait-free implementation of a k-ported test&set object from 2-consensus objects and registers, for all $k \geq 2$ [AWW93]. This, together with Claim 6.2.1, implies:

Claim 6.2.2 *There is a wait-free implementation of a $(t+1)$-consensus object from 2-consensus objects and objects of types in $\mathcal{S}$.*

We now have all the ingredients needed to complete the proof of Lemma 6.2. Since $t \geq 2$, Claim 6.2.2 implies that there is a wait-free implementation of a $(t+1)$-consensus object from t-consensus objects and objects of types in $\mathcal{S}$. This, together with Theorem 5.2, implies that there is a wait-free implementation of a $(t+1)$-consensus object from just objects of types in $\mathcal{S}$. This completes the proof of Lemma 6.2. $\square$

The following theorem, stating the close relationship between t-resilient and wait-free implementations of consensus, is immediate from Lemmata 6.1 and 6.2.

Theorem 6.3 *For all $n > t \geq 2$ and all sets $\mathcal{S}$ of types that include* `register`*, there is a t-resilient implementation of an n-consensus object from objects of types in $\mathcal{S}$ if and only if there is a wait-free implementation of a $(t+1)$-consensus object from objects of types in $\mathcal{S}$.*

7 Concluding remarks

We now present some corollaries of the equivalence theorem and briefly revisit the two formulations of t-resilience discussed in Section 2.

Herlihy proved that there is no wait-free implementation of 3-consensus from objects belonging to any of the following sets of types [Her91]: {`queue`, `register`}, {`stack`, `register`}, {`fetch&add`, `register`}, {`swap`, `register`}. (Here, the types `register`, `queue`, `stack`, `fetch&add` and `swap` can have *any* number of ports; the impossibility result holds regardless.) This, together with Lemma 6.2, implies:

Corollary 7.1 *For all $n \geq 3$, there is no 2-resilient implementation of an n-consensus object from objects belonging to any of the following sets of types:* {`queue`, `register`}, {`stack`, `register`}, {`fetch&add`, `register`}, {`swap`, `register`}.

Consider a MEM(m) object that corresponds to Herlihy's m-register assignment memory [Her91]. Informally, MEM(m) consists of an infinite array of cells; it supports the *read* i operation, which returns the value in the ith cell, and the $write(i_1, v_1, i_2, v_2, \ldots, i_m, v_m)$ operation, which (atomically) writes v_j in cell i_j, for all $j \in [1..m]$. For $m \geq 2$, Herlihy proved that there is no wait-free implementation of $(2m-1)$-consensus object from MEM(m) objects (regardless of the number of ports the MEM(m) objects may have). This, together with Lemma 6.2, implies:

Corollary 7.2 *For all $m \geq 2$ and $n \geq 2m-1$, there is no $(2m-2)$-resilient implementation of an n-consensus object from* MEM(m) *objects.*

Recall that in Section 2 we discussed two formulations of t-resilience: A strong formulation, where only access procedures that were actually invoked and left incomplete in an execution count as having crashed; and a weak formulation, where access procedures that were never invoked also count as having crashed. We then focused on the weak definition. This choice makes Lemma 6.2 stronger while making Lemma 6.1 weaker. (The other choice has the opposite effect.) The justification for this choice was that Lemma 6.2 has more interesting consequences, such as Corollaries 7.1 and 7.2, and so we wanted to make that result as strong as possible. It is worth noting, however, that Lemma 6.1 also holds under the strong definition of t-resilience. The proof of this lemma given in Section 6 does not work with this definition. To see why, consider an execution of the implementation in Figure 5 in which only one access procedure appears and is invoked at some port $i > t+1$. In this execution this procedure will never terminate, since none of the access procedures at ports $1, \ldots, t+1$ writes into R. We now sketch a proof of Lemma 6.1 under the strong definition of t-resilience. This proof uses a simulation similar to that in the proof of Lemma 6.2 (see Figure 6). The only difference is that now a larger number n of access procedures simulate the steps of a wait-free implementation for a smaller number $t+1$ of access procedures. (In Figure 6 it is the other way around: a smaller number $t+1$ of access procedures simulate the steps of a t-resilient implementation for a larger number n of access procedures.) In this way, even if only few access procedures of the t-resilient implementation are active, they will nevertheless simulate an execution of the wait-free implementation and reach a decision that satisfies validity and agreement.

Acknowledgements

We thank the anonymous referees for their comments on an earier draft of this paper.

References

[AWW93] Y. Afek, E. Weisberger, and H. Weisman. A completeness theorem for a class of synchronization objects. In *Proceedings of the 12th Annual Symposium on Principles of Distributed Computing*, pages 159–170, August 1993.

[BG93a] E. Borowsky and E. Gafni. Generalized FLP impossibility result for t-resilient asynchronous computations. In *Proceedings of the 25th ACM Symposium on Theory of Computing*, pages 91–100, May 1993.

[BG93b] E. Borowsky and E. Gafni. The implication of the Borowsky-Gafni simulation on the set consensus hierarchy. Technical Report Technical Report 930021, UCLA Computer Science Department, 1993.

[BGA94] E. Borowsky, E. Gafni, and Y. Afek. Consensus power makes (some) sense. In *Proceedings of the 13th Annual Symposium on Principles of Distributed Computing*, pages 363–372, August 1994.

[BGLR01] E. Borowsky, E. Gafni, N. Lynch, and S. Rajsbaum. The BG distributed simulation algorithm. *Distributed Computing*, 14(3):??–??, July 2001.

[BNP94] R. Bazzi, G. Neiger, and G. Peterson. On the use of registers in achieving wait-free consensus. In *Proceedings of the 13th Annual Symposium on Principles of Distributed Computing*, pages 354–363, August 1994.

[CHJT94] T. Chandra, V. Hadzilacos, P. Jayanti, and S. Toueg. Wait-freedom vs. t-resiliency and the robustness of wait-free hierarchies. In *Proceedings of the 13th Annual Symposium on Principles of Distributed Computing*, pages 334–343, August 1994.

[CIL94] B. Chor, A. Israeli, and M. Li. Wait-free consensus using asynchronous hardware. *SIAM J. on Computing*, 23(4):701–712, 1994.

[CR96] S. Chaudhuri and P. Reiners. Understanding the set consensus partial order using the Borowsky and Gafni simulation. In *Proceedings of the 10th Workshop on Distributed Algorithms*, pages 362–379, October 1996.

[DDS87] D. Dolev, C. Dwork, and L. Stockmeyer. On the minimal synchronism needed for distributed consensus. *JACM*, 34(1):77–97, 1987.

[FLP85] M. Fischer, N. Lynch, and M. Paterson. Impossibility of distributed consensus with one faulty process. *JACM*, 32(2):374–382, 1985.

[Her91] M.P. Herlihy. Wait-free synchronization. *ACM TOPLAS*, 13(1):124–149, 1991.

[HS93] M. P. Herlihy and N. Shavit. The asynchronous computability theorem for t-resilient tasks. In *Proceedings of the 25th ACM Symposium on Theory of Computing*, pages 111–120, May 1993.

[HW90] M.P. Herlihy and J.M. Wing. Linearizability: A correctness condition for concurrent objects. *ACM TOPLAS*, 12(3):463–492, 1990.

[Jay95] P. Jayanti. Wait-free computing. In *Proceedings of the 9th Workshop on Distributed Algorithms*, pages 19–50, September 1995.

[Jay97] P. Jayanti. On the robustness of Herlihy's hierarchy. *JACM*, 44(4):592–614, 1997.

[JT92] P. Jayanti and S. Toueg. Some results on the impossibility, universality, and decidability of consensus. In *Proceedings of the 6th Workshop on Distributed Algorithms*, November 1992.

[LAA87] M. Loui and H. Abu-Amara. Memory requirements for agreement among unreliable asynchronous processes. In *Advances in Computing Research*, volume 4, pages 163–183. JAI Press Inc., 1987.

[Lam77] L. Lamport. Concurrent reading and writing. *Communications of the ACM*, 20(11):806–811, 1977.

[Lam86] L. Lamport. On interprocess communication, Parts I and II. *Distributed Computing*, 1(2):77–101, 1986.

[LH00a] W.K. Lo and V. Hadzilacos. All of us are smarter than any of us: nondeterministic wait-free hierarchies are not robust. *SIAM J. on Computing*, 30(3):689–728, 2000.

[LH00b] W.K. Lo and V. Hadzilacos. On the power of shared object types to implement one-resilient consensus. *Distributed Computing*, 13(4):219–238, 2000.

[Lo95] W. K. Lo. More on t-resilience vs. wait-freedom. In *Proceedings of the 14th Annual Symposium on Principles of Distributed Computing*, pages 110–119, August 1995.

[LT88] N. Lynch and M. Tuttle. An introduction to input/output automata. Technical Report MIT/LCS/TM-373, MIT, MIT Laboratory for Computer Science, 1988.

[MR96] S. Moran and L. Rappoport. On the robustness of $\mathtt{h}_{\mathtt{m}}^{\mathtt{r}}$. In *Proceedings of the 10th Workshop on Distributed Algorithms*, pages 344–361, October 1996.

[PBN94] G. Peterson, R. Bazzi, and G. Neiger. A gap theorem for consensus types. In *Proceedings of the 13th Annual Symposium on Principles of Distributed Computing*, pages 344–353, August 1994.

[Pet83] G. L. Peterson. Concurrent reading while writing. *ACM TOPLAS*, 5(1):56–65, 1983.

[Sch96] E. Schenk. *Computability and Complexity Results for Agreement Problems in Shared Memory Distributed Systems.* PhD thesis, University of Toronto, 1996.

[SZ93] M. Saks and F. Zaharoglou. Wait-free k-set agreement is impossible: The topology of public knowledge. In *Proceedings of the 25th ACM Symposium on Theory of Computing*, pages 101–110, May 1993.