Distributed Computing manuscript No.
(will be inserted by the editor)

The Computational Structure of Progress Conditions and Shared Objects

Gadi Taubenfeld

Received: date / Accepted: date

Abstract We study the effect of different progress conditions on the computability of distributed systems. For a system with n processes, we define exponentially many new progress conditions and explore their properties and relative strength. We cover many known and new interesting conditions. Next, we propose a new classification for evaluating the strength of shared objects. The classification is based on finding, for each object of type o, the strongest progress condition for which it is possible to solve consensus for *any* number of processes, using any number of objects of type o and atomic registers. Comparing our classification with the traditional one, which is based on fixing the progress condition (namely, wait-freedom) and finding the largest number of processes for which consensus is solvable, reveals interesting results. Together with our technical results, the new definitions provide a deeper understanding of synchronization and concurrency.

Keywords Progress conditions · shared objects · wait-freedom · k-obstruction-freedom · S-freedom · consensus · synchronization · wait-free hierarchy · power hierarchy · universality

Preliminary versions of the results presented in this paper appeared in: (1) the proceedings of the 13th international conf. on principles of distributed systems (OPODIS 2009, *best paper award*) [33], and (2) the proceedings of the 24th international symposium on distributed computing (DISC 2010) [34].

G. Taubenfeld
The Interdisciplinary Center, P.O.Box 167, Herzliya 46150, Israel
E-mail: tgadi@idc.ac.il

1 Introduction

1.1 Motivation: Alice's question

Alice, the CTO of MultiBrain Inc., was excited when she told her spouse Bob about the new multi-core computer that her company has just bought. "It has 1024 cores, and it is extremely powerful," she said, "we are already writing concurrent applications that will take advantage of its high level of parallelism."

Bob, a theoretician, has asked her if they intend to use locks in their concurrent applications. "No," said Alice, "we have decided to avoid using locks." "And what are the type of atomic operations the machine supports?" asked Bob. "Swap, and reads and writes," answered Alice.[1] "What about compare-and-swap, and load-link/store-conditional?" asked Bob. "No," answered Alice, "only atomic swap, and reads and writes are supported." "Well," said Bob, "for many interesting applications you will not be able to use more than two of these 1024 cores simultaneously, so do not expect to get too much speedup." "Why?" asked Alice, "we have paid so much for that machine, it must be able to do better."

"There is a fundamental result," explained Bob, "that using the type of atomic operations that your machine supports, it is possible to solve consensus and many other important problems for two cores only, assuming that the required progress condition is wait-freedom."

Alice become serious, and asked, "the wait-freedom progress condition guarantees that a process, which corresponds to a given computation on some core, will always be able to complete its pending operations in a finite number of steps, regardless of the behavior of the other processes, right?" "Yes," answered Bob.

[1] A swap operation takes as arguments a shared register and a local register, and atomically exchange their values.

Alice laughed with relief. “You got me worried for a minute,” she said, “for all our practical purposes, we can do with a weaker progress condition than wait-freedom.” “Such as?” asked Bob. “You tell me,” answered Alice, “what is the *strongest* progress condition for which it is possible to solve consensus and all the other important problems for *any* number of cores using only atomic swap, and reads and writes?” “Interesting question,” said Bob.

1.2 Contributions and roadmap

The paper is made up of 12 sections. In Sections 2–6, the focus is on exploring the relative strength of the exponentially many progress conditions defined in Section 3. In Sections 7–9, the focus is on exploring the computational power of shared objects. Section 10 suggests two ways to extend the definitions of progress conditions. Section 11 covers the related work. Section 12 concludes the paper. Below we give a summary the results.

In the first part of the paper (Sections 2 – 6), before we help Bob resolve Alice’s question, we expand our understanding of the possible types and relative strength of different progress conditions. Below we informally define these exponentially many new progress conditions. Let n denotes the number of processes. For every non-empty set $S \subseteq \{1, ..., n\}$, the progress condition *S-freedom* guarantees that for every set of processes P where $|P| \in S$, if at some point in a computation each one of the processes in P has taken at least one step, then every process in P will be able to complete its pending operations, provided that all the processes not in P do not take steps for long enough; and that each of the processes in P with a pending operation continues to take steps.

We start our technical results by proving two general impossibility results, about the computational strength of various *S-freedom* progress conditions, which have many interesting implications (Section 4). Two such implications are:

- We uncover an interesting relationship between two important classes of progress conditions, k-obstruction-freedom, and t-resiliency, and prove that the former is stronger (Section 5). For $k \geq 1$, the progress condition *k-obstruction-freedom* guarantees that for every set of processes P where $|P| \leq k$, every process in P will be able to complete its pending operations in a finite number of steps, if all the processes not in P do not take steps for long enough time [32]. For $t \geq 1$, the *t-resiliency* condition guarantees that every process that does not fail will complete its pending operations provided that at most t processes fail.
- For the special case where only atomic registers are used, we give a complete characterization under which progress conditions the consensus problem is solvable (Section 6). This result for atomic registers generalizes the famous impossibility result for the case of one faulty process [8,25].

We notice that, using the notion of S-freedom informally defined earlier, *k-obstruction-freedom* is the same as $\{1, ..., k\}$-freedom, and *t-resiliency* is the same as $\{n-t, ..., n\}$-freedom.

In the second part of the paper (Sections 7 – 9), we propose a new classification for evaluating the strength of shared objects and compare it with the traditional one.

- The traditional classification is based on the notion of a consensus number [15]. The *consensus number* of an object of type o, denoted $CN(o)$, is the largest n for which it is possible to solve consensus for n processes, using any number of objects of type o and atomic registers, assuming that the required progress condition is *wait-freedom*. If no largest n exists, the consensus number of o is infinite. Wait-freedom, which is the same as $\{1, ..., n\}$-freedom, guarantees that every process will always be able to complete its pending operations in a finite number of steps [15].
- The new classification is based on the notion of a power number. The *power number* of an object of type o, denoted $PN(o)$, is the largest k for which it is possible to solve consensus for any number of processes, using any number of objects of type o and atomic registers, assuming that the required progress condition is *k-obstruction-freedom*. If no largest k exists, the power number of o is infinite.

Comparing our new classification with the traditional one reveals an interesting result: the two classifications are equivalent. That is, for any object of type o, $PN(o) = CN(o)$.

The results regarding the computational structure of the new and known progress conditions, provide a deeper understanding of synchronization and concurrency. Furthermore, the new classification (Section 7) together with the universality result, presented in Section 9, enables us to answer Alice’s question for *any* problem or object.

1.3 Back to Alice’s question

For the very special case of Alice’s question, we can fully answer her question:

> On Alice’s machine, which supports only atomic read/write registers and atomic swap objects, any object or problem has a 2-obstruction-free implementation for any number of processes; and this claim does not hold for 3-obstruction-freedom.

“Please explain me,” requested Bob once he understood the new results, “when you write a concurrent application for your new machine, and contention goes above two, what do you do?”

"In such cases, contention resolution schemes such as exponential backoff, are used," explained Alice, "and since with a machine which supports an atomic swap, we can easily deal with the contention of two processes, the contention resolution scheme can kick-in only once three processes are contending, and not before. Furthermore, once the contention resolution scheme kicks-in, it is enough to use it until two, and not just one, contending processes stay alive. From a practical point of view, this is a big performance gain."

"I see," said Bob, "so there is a tradeoff between how strong the progress condition should be, and how often the contention resolution scheme will kick-in." "Exactly," said Alice, "this is one of the reasons why the new classification is so helpful."

2 Computational model

Our model of computation consists of an asynchronous collection of n processes that communicate via shared objects. Asynchrony means that there is no assumption on the relative speeds of the processes.

A *process* corresponds to a given computation. That is, given some program, its execution is a process. Sometimes, for convenience, we will refer to the program code itself as a process. A process runs on a *processor* or a *core*, which is the physical hardware. Several processes can run on the same processor although in such a case only one of them may be active at any given time. Real concurrency is achieved when several processes are running simultaneously on several processors. We assume that a process may access *at most one* object at any given time.

An *object* is a memory location in which data is stored. Each object supports a set of *operations* which can be invoked by processes to manipulate that object. Each operation execution begins with an *invocation* by a process and remains *pending* until the invoking process receives a *response*. Each process may have at most one pending operation at a time. Invocations do not block: it is required that for every invocation the object can return a response, regardless of the behavior of the processes and how many other operations were invoked.

An object has a *state* which reflects its currently stored data. The *Sequential specification* of an object specifies how the object behaves when operations are applied sequentially. A *progress condition* specifies the conditions under which an operation that is invoked by a process on a given object is guaranteed to respond and terminate.

Linearizability is a consistency condition which requires that each operation should appear to take place instantaneously at some point in time and that the relative order of non-concurrent operations is preserved. For the rest of the paper, we restrict our attention only to objects which have a sequential specification, and consider only linearizable implementations.

An *event* corresponds to an atomic step performed by a process. For example, the events which correspond to accessing registers are classified into two types: read events which may not change the state of the register, and write events which update the state of a register but do not return a value. We use the notation e_p to denote an instance of an arbitrary event at a process p.

A *run* is a pair $(f,\ R)$ where f is a function which assigns initial states (values) to the objects and R is a finite or infinite sequence of events. An algorithm $Alg = (C, N, O)$ consists of a nonempty set C of runs, a set N of processes, and a set of shared objects O. An algorithm is also called an implementation. For any event e_p at a process p in any run in C, the object accessed in e_p must be in O. Let $x = (f,\ R)$ and $x' = (f',\ R')$ be runs. Run x' is a *prefix* of x (and x is an *extension* of x'), denoted $x' \leq x$, if R' is a prefix of R and $f = f'$. When $x' \leq x$, $(x - x')$ denotes the suffix of R obtained by removing R' from R. Let $R; T$ be the sequence obtained by concatenating the finite sequence R and the sequence T. Then $x; T$ is an abbreviation for $(f,\ R; T)$.

Event e_p is enabled at run x if $x; e_p$ is a run. For simplicity, we write xp to denote either $x; e_p$ when some event e_p is enabled in x, or x when no event by p is enabled in x. Register r is a *local* register of p if only p can access r. For any sequence R, let R_p be the subsequence of R containing all events in R which involve p. Runs $(f,\ R)$ and $(f',\ R')$ are *indistinguishable* for a set of processes P, denoted by $(f,\ R)[P](f',\ R')$, iff for all $p \in P$, $R_p = R'_p$ and $f(r) = f'(r)$ for every local register r of p. When $P = \{p\}$ we write $[p]$ instead of $[P]$. It is assumed that the processes are deterministic, that is, if $x; e_p$ and $x; e'_p$ are runs then $e_p = e'_p$.

The runs of an asynchronous implementation of an object must satisfy several properties: Every prefix of a run is also a run. If some event which involves p is enabled at run x, then the same event is enabled at any finite run that is indistinguishable to p from x, provided that the register p access in that event has the same value in both runs. If a *write* event which involves p is enabled at run x, then the same event is enabled at any finite run that is indistinguishable to p from x. It is possible to read only the last value that is written into a register. In an implementation of an object, an operation supported by the object, may involve (be implemented by) one or more atomic steps. Finally, for each run $x \in C$ and object $o \in O$, the subsequence of events of x which involves o, satisfies o's sequential specification.

At any time a process that intends to perform an operation can be in one of three possible distinct states (w.r.t. the operation): (1) *passive*, if it has not taken any step yet; (2) *active*, if it has taken at least one step and has not completed executing its last statement; and (3) *terminated*, if it completed executing its last statement. An active or passive process *fails* if it does not continue to (or start to) take any more steps.

3 Exponentially many progress conditions

We start our investigation by defining exponentially many progress conditions and exploring their relative strength. For a set of processes P, let $|P|$ denotes the size of P. For a given point in a computation, $active.P$ is the *number* of processes in P that are active, and $terminated.P$ is the *number* of processes in P that have terminated. We use S to denote a non-empty set such that $S \subseteq \{1, ..., n\}$.

> *Definition.* For every non-empty set $S \subseteq \{1, ..., n\}$, the progress condition *S-freedom* guarantees that for every set of processes P, if at some point in a computation $active.P + terminated.P = |P|$ and $|P| \in S$, then every process in P will be able to complete its pending operations, provided that (1) all the processes not in P do not take steps for long enough; and (2) none of the active processes in P fails (which means that each of the active processes in P will continue to take steps).

The words "long enough" are used to capture the arbitrary duration required by that process to execute the operation. Let A be an algorithm for n processes that satisfies S-freedom for some set S. Furthermore, assume that for some number k, $k \in S$ and $k - 1 \notin S$. Assume that for a set of processes P at some point in a computation of A, $active.P = |P| = k$, and that all the processes not in P do not take any more steps. As the computation progresses, assume that eventually, some process in P terminates. Once this happens $active.P = k - 1$ and $terminate.P = 1$. It is important to notice, that although at this point $active.P = k - 1 \notin S$, the definition of *$S$-freedom* guarantees that (assuming that all the processes not in P do not take any more steps) eventually every process in P will terminate, provided that no active process in P fails, because $active.P + terminated.P = |P| = k$.

Since the number of non-empty subsets of $\{1, ..., n\}$ is $2^n - 1$, there are $2^n - 1$ different progress conditions. They relate to known progress conditions as follows:

- The condition $\{n\}$-freedom is called *fault-freedom*. It guarantees that every process will complete its pending operations provided that all the n processes eventually become active and there are no failures.
- $\{1\}$-freedom is called *obstruction-freedom*. It guarantees that a process will be able to complete its pending operations in a finite number of steps, if all the other processes "hold still" (i.e., do not take any steps) long enough [16]. Obstruction-freedom does not guarantee progress under contention. (the words "long enough" are used to capture the arbitrary duration required by that process to execute the operation).
- $\{1, ..., n\}$-freedom is called *wait-freedom*. It guarantees that every process will always be able to complete its pending operations in a finite number of steps, regardless of the behavior of the other processes [15].
- $\{1, n\}$-freedom is the progress condition which implies both obstruction-freedom and fault-freedom.
- For $k \geq 1$, $\{1, ..., k\}$-freedom is called *k-obstruction-freedom*. It guarantees that for every set of processes P where $|P| \leq k$, every process in P will be able to complete its pending operations in a finite number of steps, if all the processes not in P do not take steps for long enough [31,32]. 1-obstruction-freedom is the same as obstruction-freedom.
- For $0 \leq t \leq n - 1$, $\{n - t, ..., n\}$-freedom is called *t-resiliency*. It guarantees that every process that does not fail will complete its pending operations provided that at most t processes fail.

Clearly, an object that satisfies T-freedom satisfies also S-freedom, for any $S \subset T$. For any given set S, we say that S-freedom is a *symmetric* progress condition in the sense that a given process is not favored with respect to the others.

It is possible to weaken the requirement that "*every* process in P will be able to complete its pending operations eventually", and only require that "*some* process in P will be able to complete its pending operations eventually". For one-shot objects like consensus, most of our results also apply in this case.

4 Impossibility results

The notion of a *consensus object* is central to our investigation. A consensus object o supports a single operation: $o.propose(v)$ satisfying: (1) *Agreement*: In any run, the operation $o.propose()$ returns the same value, called the *consensus value*, to every process that invokes it. (2) *Validity*: In any run, if the consensus value is v, then some process invoked $o.propose(v)$. When the set of proposed values is restricted to $\{0, 1\}$ the object is called a binary consensus object. Throughout the paper, by a consensus object we mean a binary consensus object; and by *n-valued consensus* we mean a multi-valued consensus object where $v \in \{0, 1, ..., n-1\}$.

In order to prove the impossibility results presented below, we use bivalency and covering arguments. Our proof technique is inspired and guided by the proof technique used in [25], for proving that there is no consensus object that can tolerate even a single crash failure in an asynchronous shared memory model in which only atomic registers are supported.

We use S and T to denote non-empty sets which are subsets of $\{1,...,n\}$; $|S|$ is the number of elements in S, and $max.S$ and $min.S$ are the largest and the smallest elements in S, respectively. The $width$ of S, denoted $width.S$, is defined as follows: $width.S = 1 + max.S - min.S$. Thus, the width of the set $\{1,...,n\}$ is n. We notice that it is always the case that $width.S \geq |S|$.

As we will prove below, for S-free objects, it is not $max.S$ or $min.S$ that is important for building reductions between S-free and T-free objects. What is important is the *width* of S. The intuition behind this observation can be explained as follows: Every implementation of an S-free consensus object for n processes (as we prove later), must use an object, say o, which at least $width.S$ processes must be able to access at the same time, and o is not a register. This basic fact has many immediate implications, for example, it implies that it is impossible to implement an S-free consensus object for n processes using any number of wait-free consensus objects for $width.S - 1$ processes and registers.

Theorem 1 *For any set $|S| \geq 2$, it is impossible to implement an S-free consensus object for n processes using any number of wait-free consensus objects for $width.S - 1$ processes and registers.*

It follows immediately from Theorem 1 that for any $2 \leq k \leq n$, it is impossible to implement a $\{1, k\}$-free consensus object for n processes using any number of wait-free consensus objects for $k - 1$ processes and registers. We prove in Section 8 (i.e., Algorithm 2) that when $1 \in S$, it is possible to implement an S-free consensus object for n processes using wait-free consensus objects for $width.S$ processes and registers. Thus, for any $|S| \geq 2$, the impossibility result stated in Theorem 1 is tight for S-freedom, when $1 \in S$.

Next, we consider the relative strength of different progress condition for the same number of processes.

Theorem 2 *For any two sets S and T, and integer k, if $|T| \geq 2$, $k \in T$, $k \notin S$ and $k \leq width.T$ then it is impossible to implement a T-free consensus object for n processes using any number of S-free consensus objects for n processes and registers.*

It follows from Theorem 2 that: For any $n > 2$, it is impossible to implement a wait-free consensus object for two processes using any number of $\{1, n\}$-free consensus objects for n processes and registers. Below we prove the theorems.

4.1 Detailed proofs of Theorem 1 and Theorem 2

The proofs of Theorem 1 and Theorem 2 use the following notions, abbreviations, and lemmas. Let N be the set of all n processes, and let $P \subseteq N$. A finite run x is (P, v)*-valent* if in all extensions of x, by processes in P only, where a decision is made, the decision value is v ($v \in \{0, 1\}$). A run is P*-univalent* if it is either $(P, 0)$-valent or $(P, 1)$-valent. Otherwise it is P*-bivalent*. We say that two P-univalent runs are P*-compatible* if they have the same valency, that is, either both runs are $(P, 0)$-valent or both are $(P, 1)$-valent. Finally, we say that process $p \in P$ is a P*-decider* at run x if for every extension y of x by steps of processes from P only, the run yp is P-univalent. Recall that we assume that $S \subseteq \{1,...,n\}$.

Lemma 1 *Let $|S| \geq 2$, and let P be a set of processes such that $|P| = max.S$. Then, for every $p \in P$, there is at least one subset of P, denoted $p.SP$, of size $min.S$ which does* not *include p.*

Proof From the fact that $|S| \geq 2$, it follows that $min.S < max.S$. Thus, $min.S \leq |P - \{p\}|$, and hence any subset of $P - \{p\}$ of size $min.S$ will do. □

Lemma 2 *For a set S and non-empty sets of processes P and Q such that $|P| \in S$, $|Q| \in S$ and $Q \subseteq P$, in any implementation of S-free consensus object, if two P-univalent runs are indistinguishable for Q and the state of all the objects that (processes in) Q can access are the same at these runs, then these runs must be P-compatible.*

Proof Let w and y be P-univalent runs such that $w[Q]y$, and the state of all the objects (local and shared) that processes in Q can access are the same at w and y. (See Figure 1.) Let w be (P, v)-valent, for $v \in \{0, 1\}$. Then by the definition of S-freedom, there is an extension x of w by events of Q only in which some process $p \in Q$ decides v. Clearly $z = y; (x - w)$ is also a run of the algorithm such that $z[Q]x$. Since p decides v in z, z is (P, v)-valent. Hence, since $y \leq z$, y must also be (P, v)-valent. □

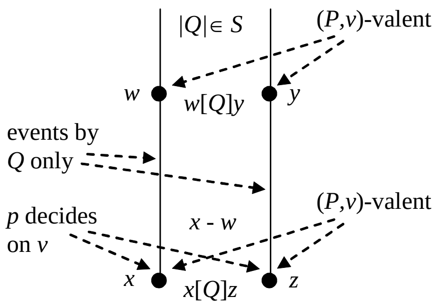

Fig. 1 Illustration of runs in the proof of Lemma 2.

Lemma 3 *Let $|S| \geq 2$ and let P be a set of processes such that $|P| = max.S$. Then, every implementation of S-free consensus object has a P-bivalent empty run.*

Proof We show that a P-bivalent empty run must exist. Assume to the contrary that every empty run is P-univalent. The empty run with all 0 inputs must be $(P, 0)$-valent, and similarly the empty run with all 1 inputs must be $(P, 1)$-valent. Let $Q \subset P$ be a set of processes such that $|Q| = min.S$. Clearly, all the empty runs in which all the processes has 0 inputs, and all the empty runs in which all the processes has 0 inputs, except for the input of one process in $P - Q$, are indistinguishable for Q. Furthermore, the state of all the objects that (processes in) Q can access are the same at these runs. Thus, by Lemma 2 and the validity requirement, all the empty runs with all 0 inputs, except for the input of one process in $P - Q$, are $(P, 0)$-valent, and similarly all the empty runs with all 1 inputs, except for the input of one process in $P - Q$, are $(P, 1)$-valent. By repeatedly applying this argument $i \leq max.S/2$ times (each time choosing a new $Q \subset P$ of size $min.S$ for which the inputs do not change), we get that, all the empty runs with all but i 0 inputs for i processes in P are $(P, 0)$-valent, and similarly all the empty runs with all but i 1 inputs for i processes in P are $(P, 1)$-valent. Thus, when i is $max.S/2$, we get that there are two empty runs x_0 and x_1 that for some $p \in P$, differ at the input value of p, and agree on the input values of all the processes in $P - \{p\}$, such that x_0 is $(P, 0)$-valent and x_1 is $(P, 1)$-valent. However, this contradicts Lemma 2, when applied to x_0 and x_1 and a set $Q \subseteq P - \{p\}$ of size $min.S$. Hence, an empty P-bivalent run exists. □

Lemma 4 *Let $|S| \geq 2$ and let P be a set of processes such that $|P| = max.S$. Let y be a run of an S-free consensus object, and let $p \in P$ and $q \in P$ be two different processes such that (1) $y \neq yp$ and $y \neq yq$, (2) the runs yp and yqp are P-univalent and not P-compatible. Then, in their two next events from y, p and q are accessing the same object, and this object is not a register.*

Proof We first consider the following three possible cases, and show that each one of them leads to a contradiction. (See Figure 2.) We will assume that in the last event in yp process p is accessing some object, say o, and in the last event in yq process q is accessing some object, say o'. Recall, that by Lemma 1, for every process $p \in P$, there is at least one subset of P, denoted $p.SP$, of size $min.S$ which does *not* include p.

Case 1: $o \neq o'$. (See Figure 2(a).) Since the two next events from y of p and q are independent, $ypq[p.SP]yqp$ and the values of all objects are the same in both ypq and yqp. (Actually, $ypq[N]yqp$ where N is the set of all the n processes.) Since ypq is an extension of yp and yp is P-univalent, it follows that also ypq is P-univalent. By Lemma 2, ypq and yqp are P-compatible; and thus, since ypq is an extension of yp, it must be that yp and yqp are also P-compatible. A contradiction.

Case 2: $o = o'$ and in yp the last event is a *write* event by p to o. (See Figure 2(b).) Since p *writes* to o in its next operation from y, the value of o must be the same in yp and yqp. (Here we use the fact that the write by p overwrites the possible changes of o made by q.) Hence, $yp[q.SP]yqp$ and the values of all the objects, which are not local to q, are the same in yp and yqp. By Lemma 2, yp and yqp are P-compatible. A contradiction.

Case 3: In yp the last event is a *read* event by p. (See Figure 2(c).) Thus, $ypq[p.SP]yqp$, and the values of all the objects, which are not local to p, are the same in both ypq and yqp. By Lemma 2, ypq and yqp are P-compatible. Since ypq is an extension of yp, it must be that yp and yqp are also P-compatible. A contradiction.

Thus, it must be the case that $o = o'$ and o is not a register. □

Lemma 5 *Let $|S| \geq 2$ and let P be a set of processes such that $|P| = max.S$. For every S-free consensus object there is a P-bivalent run x and process $p \in P$ such that p is a P-decider at x.*

Proof Let $Cons$ be an arbitrary S-free consensus object. We assume w.l.o.g. that the processes in P are named $p_0, ..., p_{|P|-1}$. By $\overline{P}$ we denote the set of all processes excluding the processes in P. By Lemma 3, $Cons$ has an empty P-bivalent run x_0. We begin with x_0 and pursue the following round-robin *P-bivalence-preserving scheduling* discipline:

```
1 $x := x_0$; $Q := \emptyset$; $i := 0$          //initialization
2 repeat
3     if $x$ has a $P$-bivalent extension $yp_i$ where $x[\overline{P}]y$
4     then $x := yp_i$   //$P$-bivalent ext. of $x$
5     else $Q := p_i$ //no such $P$-bivalent ext
6     $i := i + 1 (\mathrm{mod}\ |P|)$           //round-robin
7 until $|Q| = 1$.
```

Since $Cons$ satisfies S-freedom and $|P| \in S$, by definition the above procedure must terminate, and it will terminate with some P-bivalent finite run x, and a singleton set $Q = \{p\}$ for some process p, such that p is a P-decider at x. □

Lemma 6 *Let $|S| \geq 2$ and let P be a set of processes such that $|P| = max.S$. Every S-free consensus object has a P-bivalent run y and two processes $p \in P$ and $q \in P$ such that: (1) p is a P-decider at y; (2) the runs yp and yqp are P-univalent and not P-compatible; and (3) in their two next events from y, p and q are accessing the same object, and this object is not a register.*

Proof Let $Cons$ be an arbitrary S-free consensus. By Lemma 5, there is a process $p \in P$ and a P-bivalent run x of $Cons$ such that p is a P-decider at x.

Let $\overline{v} = 1 - v$. Suppose that the run xp is (P, v)-valent. Since x is P-bivalent, there is a (shortest) extension z of x, by event of processes in P only, which is $(P, \overline{v})$-valent.

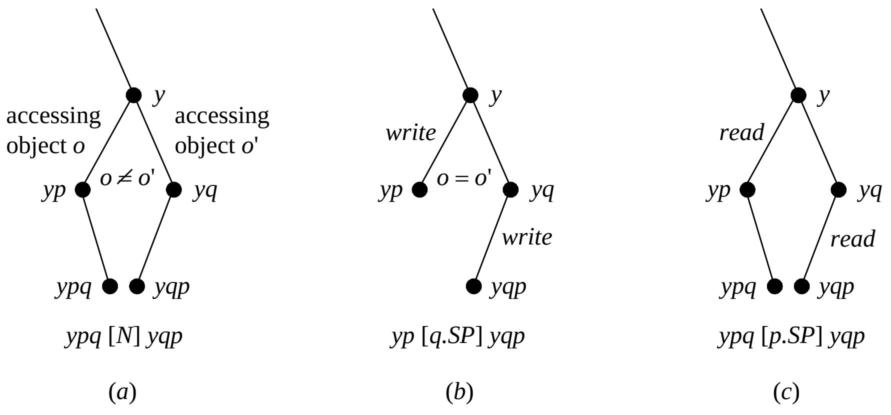

Fig. 2 Illustration of runs in the proof of Lemma 4.

(See Figure 3(a).) Let z' be the longest prefix of z such that $x[p]z'$. There are two possible cases: either (1) z' is P-univalent, in which case $z' = z$, or (2) z' is P-bivalent, in which case $z'p = z$. In both these cases, from the assumption that z is $(P, \overline{v})$-valent, it follows that $z'p$ is $(P, \overline{v})$-valent. (See Figure 3(b).)

Consider the extensions of x which are also prefixes of z'. Since $x[p]z'$ and $z' - x$ involves only events by processes in $P - \{p\}$, it follows that for every y such that $x \leq y \leq z'$, $y \neq yp$. Since xp and $z'p$ are not P-compatible, there must exist *different* runs y and yq such that (1) $x \leq y < yq \leq z'$, and $p \neq q$; (2) yp and yqp are P-univalent but not P-compatible, and (3) by Lemma 4, in their two next events from y, p and q are accessing the same object, and this object is not a register. (See Figure 3(c).) □

Lemma 7 *Let $|S| \geq 2$ and let P be a set of processes such that $|P| = max.S$. Every S-free consensus object has a P-bivalent run y, a set $Q \subseteq P$ of size $width.S$, and two processes $p \in Q$ and $q \in Q$ such that: (1) p is a P-decider at y; (2) the runs yp and yqp are P-univalent and not P-compatible; and (3) in their next events from y, all the $width.S$ processes in Q, are accessing the same object, and this object is not a register.*

Proof The proof is by induction on the number of processes k, where $2 \leq k \leq width.S$. The base of the induction follows directly from Lemma 6. We assume that the theorem holds for $k < width.S$ processes and prove for $k + 1$.

Induction hypothesis: Every S-free consensus object for n processes has a P-bivalent run x and two processes $p \in P$ and $q \in P$ such that: (1) p is a P-decider at x; (2) the runs xp and xqp are P-univalent and not P-compatible; and (3) in their next events from x, $k < width.S$ processes, including p and q, are accessing the same object, and this object is not a register. We denote by Q the set of these k processes, and assume that p and q are in Q. Since $|Q| < width.S$, $|P| - |Q| \geq min.S$.

Induction step. Let x be the run mentioned in the induction hypothesis, and let R be a set of processes such that $R \subseteq P - Q$ and $|R| = min.S$. To prove that the claim hold for $k + 1$ processes, we will show that there is a P-bivalent extension y of x by steps of processes from R only such (1) p is a P-decider at y; (2) the runs yp and yqp are P-univalent and not P-compatible; and (3) there is a process $r \in R$, such that in their next events from y, the ***k*+1** processes in $Q \cup \{r\}$ are accessing the same object, say o, and this object is not a register.

Let $\overline{v} = 1 - v$. Suppose that the run xp is (P, v)-valent and the run xqp is $(P, \overline{v})$-valent. Since x is P-bivalent and $|R| \in S$, there is a (shortest) extension z of x by steps of processes in R only which is P-univalent. (See Figure 4(a).) We first prove that in at least one of the events in $(z - x)$ some process in R is accessing o. Assume to the contrary that none of the events in $(z - x)$ involves accessing o (and recall that N is the set of all the processes). In such a case, since in their two next events from x, p and q are accessing o, we get that:

1. $xp; (z-x)[N]zp$ and the state of all the objects in $xp; (z-x)$ and zp are the same. Since $xp; (z - x)$ is an extension of xp by steps of processes in P only, it follows from the fact that xp is (P, v)-valent, that also $xp; (z - x)$ is (P, v)-valent. Since z is P-univalent, also zp is P-univalent. By Lemma 2, $xp; (z - x)$ and zp are P-compatible, and hence zp is (P, v)-valent. (See Figure 4(b).)
2. $xqp; (z - x)[N]zqp$ and the state of all the objects in $xqp; (z - x)$ and zqp are the same. Since $xqp; (z - x)$ is an extension of xqp by steps of processes in P only, it follows from the fact that xqp is $(P, \overline{v})$-valent, that also $xp; (z - x)$ is $(P, \overline{v})$-valent. Since z is P-univalent, also zqp is P-univalent. By Lemma 2, $xqp; (z - x)$ and zqp are P-compatible, and hence zqp is $(P, \overline{v})$-valent. (See Figure 4(c).)

Thus, zp and zqp are not P-compatible. But this is not possible given that zp and zqp are extensions of the *P-univalent*

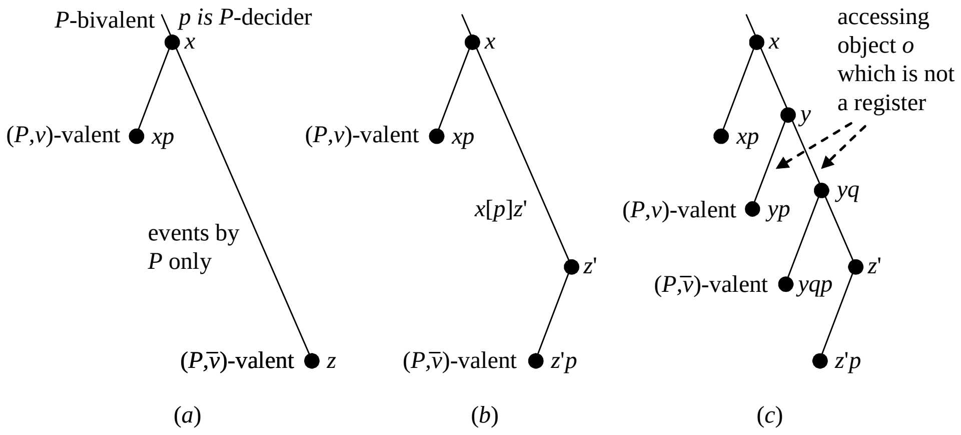

Fig. 3 Illustration of runs in the proof of Lemma 6.

run z. A contradiction. Hence, at least one of the events in $(z - x)$ access o.

Let process $r \in R$, be the first process to access o in $(z - x)$, and let $y \geq x$ be the longest prefix of z such none of the events in $(y - x)$ access o. (See Figure 5(a).) Since $yr \leq z$, y is P-bivalent. Furthermore, in its next events from y, process r is accessing o, and also in their next events from y, the ***k*** processes in Q are accessing o. Since in their two next events from x, p and q are accessing o and in $(y - x)$ no process in R is accessing o, we get that:

1. $xp; (y-x)[N]yp$ and the state of all the objects in $xp; (y-x)$ and yp are the same. Since $xp; (y-x)$ is an extension of xp by steps of processes in P only, it follows from the fact that xp is (P, v)-valent, that also $xp; (y - x)$ is (P, v)-valent. Since p is a P-decider at x, clearly yp is P-univalent. By Lemma 2, $xp; (y - x)$ and yp are P-compatible, and hence yp is (P, v)-valent. (See Figure 5(b).)
2. $xqp; (y - x)[N]yqp$ and the state of all the objects in $xqp; (y - x)$ and yqp are the same. Since $xqp; (y - x)$ is an extension of xqp by steps of processes in P only, it follows from the fact that xqp is $(P, \overline{v})$-valent, that also $xp; (y - x)$ is $(P, \overline{v})$-valent. Since p is a P-decider at x, clearly yqp is P-univalent. By Lemma 2, $xqp; (y - x)$ and yqp are P-compatible, and hence yqp is $(P, \overline{v})$-valent. (See Figure 5(b).)

Thus, as required, yp and yqp are P-univalent but not P-compatible. Finally, since p is a P-decider at x, and y is an extension of x by event of processes in $R \subset P$ only, p is also a P-decider at y. □

Proof of Theorem 1. It follows from Lemma 7 that every implementation of an S-free consensus object for n processes, must use an object, say o, which at least $width.S$ processes must be able to access at the same run, and o is not a register. Thus, it is not possible to implement an S-free consensus object for n processes using any number of wait-free consensus objects for $width.S - 1$ processes and registers. □

Proof of Theorem 2. Assume to the contrary that there is such an implementation of a T-free consensus object for n processes from S-free consensus objects for n processes and registers. Let P be a set of processes such that $|P| = max.T$. It follows from Lemma 7 that such an implementation has a P-bivalent run y, a set $Q \subseteq P$ of size $width.T$, and two processes $p \in Q$ and $q \in Q$ such that: (1) p is a P-decider at y; (2) the runs yp and yqp are P-univalent and not P-compatible; and (3) in their next events from y, all the $width.S$ processes in Q, are accessing the same object, say o, and this object is not a register. Thus, it must be the case that o is an S-free consensus object.

We notice that since y is P-bivalent no process has yet terminated in y. Assume that at the end of y, just before the $width.T$ processes access o, $n - k$ processes fail and the remaining $k \geq 1$ active processes are about to access o. Since there are only $k \geq 1$ active processes, (no process has terminated) and $k \in T$, the implementation of a T-free consensus object must guarantee that these k correct active processes will eventually properly terminate. However, since $k \notin S$, the S-free consensus object o does not guarantee that any of the remaining k active processes will ever get a response from o. Thus, the k processes will never be able to terminate. A contradiction. □

4.2 An observation

In the context of wait-freedom, it is obvious that any wait-free object for n processes is also a wait-free algorithm for $n - 1$ processes. Below, we generalize this observation.

Theorem 3 *Let o be an object for n processes that satisfies S-freedom. Then, o is also an object for $n - 1$ processes that satisfies $(S \cap \{1, ..., n - 1\})$-freedom.*

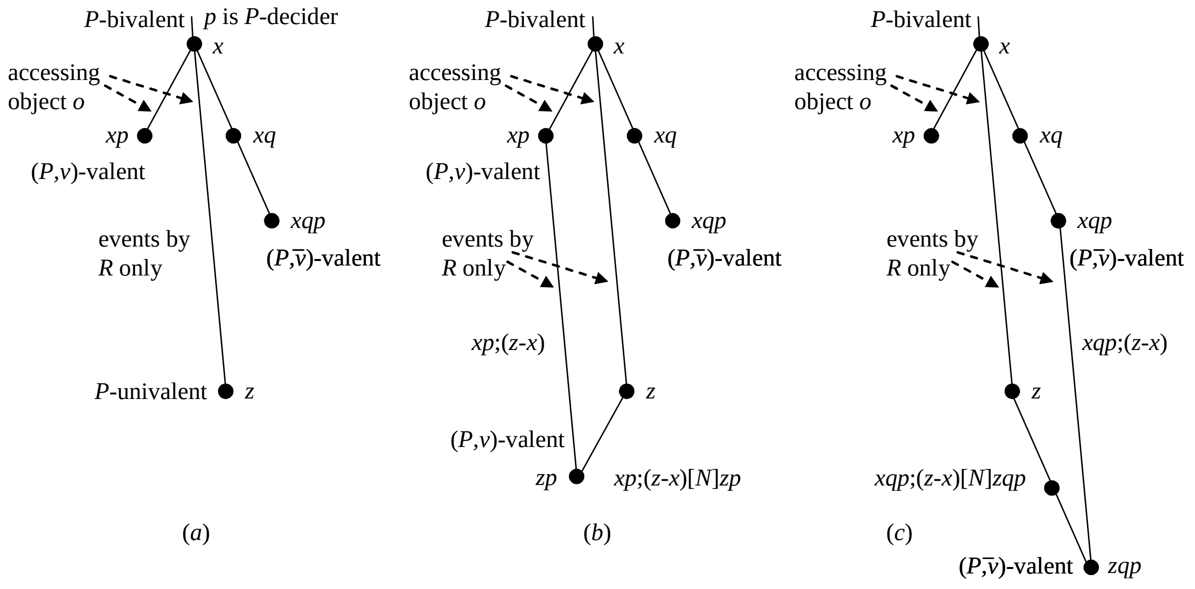

Fig. 4 Illustration of runs in the proof of Lemma 7.

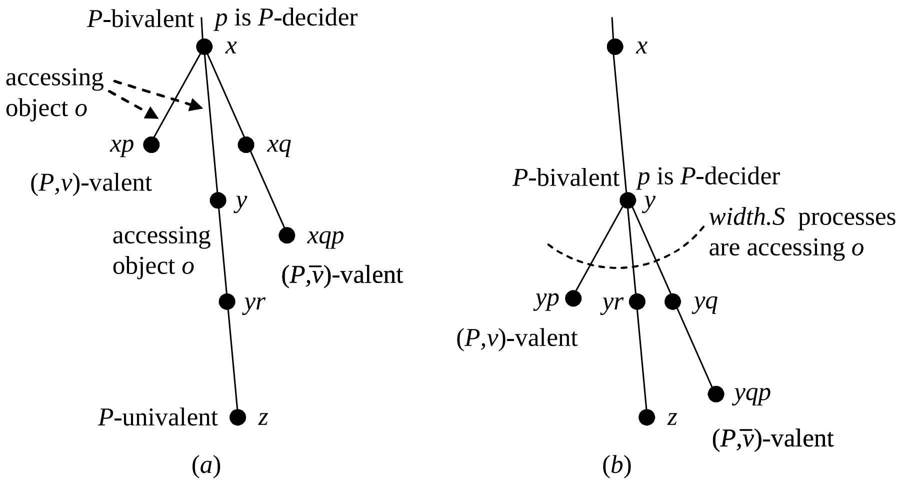

Fig. 5 Additional illustration of runs in the proof of Lemma 7.

Proof Removing some process and considering only the remaining $n-1$ processes, only affect the guarantees given for the case where all the n processes participate and none of them fails. So this case should be omitted as done in the statement of Theorem 3. □

It follows trivially from Theorem 3 that, for any $n \geq k$, a k-obstruction-free consensus object for n processes, is also k-obstruction-free consensus object for k processes. As for the other more interesting direction, it follows from a result we prove in Section 8 (i.e., Algorithm 2) that for any finite or infinite number of $n \geq k$ processes, it is possible to implement a k-obstruction-free consensus object for n processes, from wait-free consensus objects for k processes and registers.

5 *k*-obstruction-freedom vs. *t*-resiliency

We examine the relationship between the k-obstruction-freedom progress conditions and the t-resiliency progress conditions. As we show below, the k-obstruction-freedom conditions are stronger.

Theorem 4 *It is impossible to implement $\{1,2\}$-free consensus object for n processes using $\{2,...,n\}$-free consensus objects for n processes and registers.*

Proof Let T be the set $\{1,2\}$, and $k=1$. Then, (1) $|T| \geq 2$, (2) $k \in T$, (3) $k \notin \{2,...,n\}$, and (4) $k \leq width.T$. Thus, the result follows from Theorem 2. □

Next we show that consensus objects that satisfy t-resiliency conditions can be implemented from consensus objects that satisfy the corresponding k-obstruction-freedom conditions.

Theorem 5 *For $2 \leq k \leq n$, it is possible to implement an $\{n-k+1, ..., n\}$-free consensus object for n processes using $\{1, ..., k\}$-free consensus objects for n processes and registers.*

To prove Theorem 5, we first generalize a known result for wait-freedom, namely, that multi-valued consensus can be implemented from binary consensus ([30], page 329).

Lemma 8 *For any $k \geq 2$, $n \geq 2$ and $S \subseteq \{1, ..., n\}$, an S-free k-valued consensus object for n processes can be implemented from S-free binary consensus objects for n processes and atomic bits.*

Proof To implement a single k-valued consensus object, we use $\lceil \log k \rceil$ binary consensus objects, which are numbered 0 through $\lceil \log k \rceil - 1$, and k bits which are numbered 0 through $k-1$ and are initialized to 0. To propose a value $v \in \{0, ..., k-1\}$, p does the following: (1) it sets the bit number v to 1; (2) it proposes the binary encoding of v, bit by bit, to the binary consensus objects in an increasing order starting from number 0. If at some point during the second step the bit p has proposed is not accepted as the consensus value at the corresponding binary consensus object, p stops proposing v, scan the bits and chooses one of the bits that are set to 1, say v', which also matches the values that has successfully proposed so far and continues to propose the value v'. This procedure continues until p proposes, to all the $\lceil \log k \rceil$ binary consensus objects. The value that its binary encoding was successfully proposed to all the $\lceil \log k \rceil$ binary consensus objects is the final consensus value. $\square$

Proof of Theorem 5: Build a tree of degree k with $\lceil n/k \rceil$ leaves, and where each node of the tree is a $\{1, ..., k\}$-free k-valued consensus object. Each participating process is progressing from a leaf towards the root, where at each level of the tree it accesses a k-valued consensus object, competing against at most $k-1$ processes in its neighbor's subtree. As a process advances towards the root, it plays the role of process 0 (i.e., proposes 0) when it arrives from the leftmost subtree, of process $k-1$ when it arrives from the righmost subtree, or of process $0 \leq i \leq k-1$ when it arrives from the i'th subtree. The winner at each node is the process its value is being agreed upon. Only a winner at a given node continues to progress towards the root. The value agreed at the root is the final decision value. Each of the processes that accesses the root writes the final decision value at a special register called *decision*, and decides on that value. Each process that loses at some node other than the root, spins on the *decision* register until a value is written into it and decides on that value. $\square$

We point out that the construction in the above proof, of $\{n-k+1, ..., n\}$-free consensus object using $\{1, ..., k\}$-free consensus objects and registers, does not satisfy $\{1, ..., k\}$-freedom. The construction is *symmetric* in the sense that processes execute similar programs, and each value proposed by a process has a chance to be decided upon. There is a simple asymmetric construction: since at most $k-1$ processes are faulty, we can let the k processes p_1 to p_k access a designated $\{1, ..., k\}$-free consensus object. Eventually, one of them will get a response and write it in a *decision* register. The remaining processes read the decision register until a value is written into it and decide on that value.

6 Atomic registers

For the case where only registers are used, we present a complete characterization under which progress conditions the consensus problem is solvable.

Theorem 6 *For any set S, it is possible to implement an S-free consensus object for n processes using registers if and only if $|S| = 1$.*

Proof It follows from Theorem 1 that it is impossible to implement an S-free consensus object for n processes using registers if $|S| \geq 2$. We show that for any integer $1 \leq k \leq n$, it is possible to implement a $\{k\}$-free consensus object for n processes using registers. The algorithm is discribed in Figure 6.

The algorithm (i.e., implementation) proceeds in rounds. The notion of a *round* is used *only* for the sake of describing the algorithm. That is, we do *not* assume a synchronous model of execution in which all the processes are always executing the same round. and where no process can move to the next round before all others have finished the previous round.

Each process has a preference for the decision value in each round; initially this preference is the input value of the process. If no decision is made in a round then the processes advance to the next round, and try again to reach agreement.

In round $r \geq 1$, process p_i first checks if the bit of its preference v_i and of the opposite value $1-v_i$ are set. If both bits are not set, p_i sets its preference bit v_i by writing 1 to $x[r, v_i]$ (line 2). Then, p_i sets its participation bit by writing 1 to *flag*$[r_i, i]$ (line 3). Next, p_i reads the bit $x[r-1, 1-v_i]$. If the bit $x[r-1, 1-v_i]$ is not set, then every process that reaches round r with the conflicting preference $1-v_i$ will find that only $x[r, v_i]$ is set to 1, will never set $x[r, 1-v_i]$ to 1. Consequently, process p_i can safely decide on v_i, and it writes v_i to *decide* (line 4). Otherwise, waits until it notices that at least k processes are participating in round r (lines 5–8). After that p_i updates its preference in an attempt to agree with the other processes (line 9). Then, p_i proceeds to round $r+1$ (line 11).

Algorithm 1. IMPLEMENTING $\{k\}$-FREE CONSENSUS FOR n PROCESSES USING REGISTERS (WHERE $k \in \{1,...,n\}$):
`program for process` p_i `with input` in_i `(`$in_i \in \{0,1\}$ `and` $i \in \{1,...,n\}$`)`.

```
shared registers
$x[0..\infty, 0..1]$ infinite array of bits, initially $x[0,0] = x[0,1] = 1$ and all other entries are 0
$flag[1..\infty, 1..n]$ infinite array of bits, initially all entries are 0
$decide$ ranges over $\{\perp, 0, 1\}$, initially $\perp$

local registers
$r_i$ integer, initially 1
$v_i$ bit, initially $in_i$ ; $l_i$,$count_i$ integers, initial values are immaterial

1  while $decide = \perp$ do
2      if $x[r_i, 0] = 0$ and $x[r_i, 1] = 0$ then $x[r_i, v_i] := 1$ fi           // preferred value
3      $flag[r_i, i] := 1$                                                      // signal participation
4      if $x[r_i - 1, 1 - v_i] = 0$ then $decide := v_i$                          // no conflict in $r_i - 1$
5          else repeat                                                          // $k$-barrier
6                  $count_i = 0$                                                // initialize local counter
7                  for $l_i = 1$ to $n$ do if $flag[r_i, l_i] = 1$ then $count_i := count_i + 1$ fi od
8              until ($count_i \geq k$)                                         // at least $k$ participate
9              if $x[r_i, 0] = 1$ then $v_i := 0$ else $v_i := 1$ fi            // value for $r_i + 1$
10     fi
11     $r_i := r_i + 1$
12 od
13 decide($decide$)
```

Fig. 6 A $\{k\}$-free consensus algorithm for n processes using registers (where $k \in \{1,...,n\}$).

If *exactly* k processes with possibly conflicting preferences participate in round r, then they will reach line 9, only *after* all of them set their flags in line 3. This implies that once some process reaches line 9, no process is at line 2, and hence all the k processes will reach round $r+1$ with the same preference which is the value chosen in line 9. When all processes reach a round with the same preference, a decision is reached either in that round or the next round. □

7 A new classification

The traditional approach to measure the relative computational power of shared objects is to classify them according to their consensus numbers. Objects with higher numbers are considered stronger. We propose a new classification which is based on finding the strongest progress condition for which it is possible to solve consensus for *any* number of processes. The new classification together with our technical results enables us to answer Alice's question for any type of problem or object. Comparing our classification with the traditional one reveals interesting results.

7.1 The traditional approach: Consensus numbers and the wait-free hierarchy

In Section 4, we have defined the notion of a consensus object. It is sometimes convenient to use the notion of a consensus algorithm instead of a consensus object (the two notions are essentially the same). The (binary) consensus problem is to design an algorithm in which all correct processes reach a common decision based on their initial opinions. The problem is defined as follows. There are n processes $p_1, p_2, \ldots, p_n$. Each process p_i has an input value $x_i \in \{0,1\}$. The requirements of the consensus problem are that there exists a *decision value* v such that: (1) each non-faulty process eventually decides on v, and (2) $v \in \{x_1, x_2, \ldots, x_n\}$. In particular, if all input values are the same, then that value must be the decision value.

The *consensus number* of an object of type o, denoted $CN(o)$, is the largest n for which it is possible to solve consensus for n processes, using any number of objects of type o and atomic registers, assuming that the required progress condition is *wait-freedom*. If no largest n exists, the consensus number of o is infinite. The *wait-free hierarchy* is an infinite hierarchy of objects such that the objects at level i of the hierarchy are exactly those objects with consensus number i [15].

In the wait-free hierarchy, for any positive i, in a system with i processes: (1) no object at level less than i together with atomic registers can implement any object at level i; and (2) each object at level i together with atomic registers can implement any object at level i or at a lower level [15]. If instead of wait-freedom, only obstruction-freedom is required, the hierarchy collapses (which means that with obstruction-freedom the hierarchy has only one level – every object can be implemented in an obstruction-free manner using registers).

Recall that sequential specification specifies how the object behaves when operations are applied sequentially. Linearizability implies that each operation should appear to take place instantaneously at some point in time and that the relative order of non-concurrent operations is preserved [19].

An object o is *universal* for n processes if any object which has sequential specification has a wait-free linearizable implementation using atomic registers and objects of type o in a system with at most n processes. In [15], it was shown that consensus for n processes is universal in a system with n processes, for any positive n. An immediate implication of this result is that an object o is universal in a system with n processes if and only if the consensus number of o is at least n.

7.2 A new approach: Power numbers and the power hierarchy

Instead of the traditional classification of fixing a single progress condition (namely, wait-freedom) and finding the largest number of processes for which consensus is solvable, we propose a new classification which is based on finding the strongest progress condition for which it is possible to solve consensus for *any* number of processes. We restrict our attention to the progress conditions k-obstruction-freedom. These conditions cover the spectrum between obstruction-freedom and wait-freedom. When the maximum number of processes is a fixed number, say n, k-obstruction-freedom is the same as wait-freedom, for all $k \geq n$.

Definition. The *power number* of an object of type o, denoted $PN(o)$, is the largest k for which it is possible to solve consensus for any number of processes, using any number of objects of type o and atomic registers, assuming that the required progress condition is *k-obstruction-freedom*. If no largest k exists, the power number of o is infinite. The *power hierarchy* is an infinite hierarchy of objects such that the objects at level i of the hierarchy are exactly those objects with power number i.

In the above definitions, the objects of type o and the atomic registers are all assumed to be wait-free. That is each operation that is invoked by a process on these basic objects always terminates, regardless of the behavior of the other processes. [2]

Next we generalize the notion of universality which was defined earlier in the context of the wait-free hierarchy.

Definition. An object o is *k-universal* if any object which has sequential specification has a *k-obstruction-free* linearizable implementation using atomic registers and objects of type o for any number of processes.

Clearly, if an object is k-universal its is also universal for k processes. An interesting question that we resolve in Section 9 is whether an object that is universal for k processes is k-universal in a system with *more than* k processes. The power hierarchy is meaningful because it is defined using only the progress conditions $\{1, ..., k\}$-freedom, for all k. In such a case, there is a total order, based on the "stronger than" relation, between all these conditions. The result from Section 5 supports the use of k-obstruction-freedom over t-resiliency for defining the power hierarchy.

8 An equivalence result

We show that the wait-free hierarchy and power hierarchy are equivalent. Put another way, we show that the consensus number of an object equals to its power number.

Theorem 7 (Equivalence) *For any object of type o, $PN(o) = CN(o)$.*

Proof In the sequel, the term k-obstruction-free consensus algorithm, means a consensus algorithm that satisfies k-obstruction-freedom for *any* number of processes. It follows immediately from the definitions that, for any $k \geq 1$, a k-obstruction-free consensus algorithm is also a wait-free consensus algorithm for k processes. This simple observation implies that, for any object type o, $PN(o) \leq CN(o)$. The difficult part of the proof is to show that, for any $k \geq 1$, it is possible to implement a k-obstruction-free consensus algorithm using only wait-free consensus objects for k processes and atomic read/write registers. Such an implementation together with the above observation implies the theorem.

In Figure 7, a an implementation of k-obstruction-free consensus algorithm is present with a correctness proof. In the implementation, we use a function, called $set_k()$, from the positive integers into sets of size k of process ids. We do not care how exactly the function is implemented, we only

[2] We notice that k-obstruction-freedom defined for a distinct system size is a distinct progress condition. By saying that it possible to solve consensus for any number of processes, assuming that the required progress condition is k-obstruction-freedom, we mean that it is possible to do so for any fixed number of processes n assuming that the required progress condition is k-obstruction-freedom for n processes.

care that, for every positive integer k, there exists a function $set_k()$ which satisfies the following property: For every set of k process ids P and every positive integer s there exists $t \geq s$ such that $P = set_k(t)$. That is, every set of k process ids appears infinitely often.

The algorithm proceeds in rounds. The notion of a *round* is used only for the sake of describing the algorithm. We do *not* assume a synchronous model of execution in which all the processes are always executing the same round, and where no process can move to the next round before all others have finished the previous round. Each process has a preference for the decision value in each round; initially this preference is the input value of the process. If no decision is made in a round then the processes advance to the next round, and try again to reach agreement.

In round $r \geq 1$, process p_i first checks if the flag of its preference v_i is already set. If it is not set and the flag of $1 - v_i$ is set, p_i changes its preference to $1 - v_i$. If both flags are not set, p_i flags its preference v_i by writing 1 to $x[r, v_i]$ (line 2). Then, p_i reads the flag $x[r - 1, 1 - v_i]$. If the flag $x[r - 1, 1 - v_i]$ is not set, then every process that reaches round r with the conflicting preference $1 - v_i$ will find that only $x[r, v_i]$ is set to 1, and will change its preference to v_i. Consequently, process p_i can safely decide on v_i, and it writes v_i to *decide* (line 3). Otherwise, p_i checks if it belongs to the set $set_k(r)$. If it does, p_i proposes its current preference v_i to $con[r]$ and updates its preference to be the value agreed upon in $con[r]$. Then, p_i proceeds to round $r + 1$ (line 4).

If only up to k processes with conflicting preferences participate in round r, and all of them are in $set_k(r)$, then all of them will reach round $r + 1$ with the same preference which is the value agreed upon in $con[r]$. When all processes reach a round with the same preference, a decision is reached either in that round or the next round.

Theorem 8 *Algorithm 2 is a correct k-obstruction-free consensus algorithm for any number of processes, using atomic registers and wait-free consensus objects for k processes.*

Before we prove the theorem, we make the following observations:

- Let Algorithm 2.1 be a modified version of Algorithm 2, where line 4 is omitted. Then, Algorithm 2.1 is a correct 1-obstruction-free consensus algorithm for any number of processes using atomic registers only.
- Let $y[1..\infty]$ be an infinite array of swap objects, which range over $\{\perp, 0, 1\}$, initially all set to $\perp$, and let $temp_i$ be a local register of process p_i. Let Algorithm 2.2 be a modified version of Algorithm 2, where line 4 is replaced with:
  "**else** $temp_i := v_i$; $swap(y[i], temp_i)$; **if** $temp_i \neq \perp$ **then** $v_i := temp_i$ **fi**"
  Then, Algorithm 2.2 is a correct 2-obstruction-free consensus algorithm for any number of processes using atomic registers and swap objects.
- Let $y[1..\infty]$ be an infinite array of test&set bits, initially all set to 0, and let Algorithm 2.3 be a modified version of Algorithm 2, where line 4 is replaced with:
  "**else if** $test\&set(y[r_i]) = 1$ **then if** $x[r_i, 1 - v_i] = 1$ **then** $v_i := 1 - v_i$ **fi fi**"
  Then, Algorithm 2.3 is a correct 2-obstruction-free consensus algorithm for any number of processes using atomic registers and test&set bits[3].

Below we present a correctness proof of the algorithm. Let $r \geq 1$ and $v \in \{0, 1\}$. Process p_i *reaches* round r, if it executes Statement 2 with $r_i = r$. Process p_i *prefers* the value v in round r, if $v_i = v$ when p_i reaches round r. Process p_i *commits* to the value v in round r, if it executes the assignment *decide* $:= v$ with $r_i = r$.

Lemma 9 *If all processes reaching round r have the same preference v for round r, then all nonfaulty processes reaching round r commit to v either in round r or in round $r + 1$.*

Proof Suppose all processes reaching round r have the same preference v for round r. Then, whenever some process p_i sets the bit $x[r, v_i]$ to 1, v_i equals v. Consequently, no process ever sets $x[r, 1 - v]$ to 1, and hence $x[r, 1 - v]$ always equals 0. Consider a process p reaching round r. Assuming that p continues to take steps in round r, p will either (1) finds $x[r - 1, 1 - v] = 0$ at Statement 3, and commits to the value v in round r, or (2) will continue to round $r + 1$ with preference v. In the second case, all processes reaching round $r+1$ have the same preference v for round $r+1$, thus, p will find $x[r, 1 - v] = 0$ at round $r + 1$, and will commit to v in round $r + 1$. □

Lemma 10 (validity) *If p_i decides on a value v then $in_j = v$ for some p_j.*

Proof If there are two processes that have different inputs then the lemma holds trivially. Suppose all processes start with the same input *in*. Then, by Lemma 9, all nonfaulty processes will commit to in in the first round or the second round (actually the second round in this case), will execute the statement **decide**(*decide*), at the end of one of these two rounds, and will decide on *in*. □

Lemma 11 *If some process commits to v in round r then all processes reaching round $r + 1$ prefer v and commit to v in round $r + 1$.*

[3] A test&set bit, say r, is a shared bit that supports two operations: (1) $test\&set$, which writes 1 to r, and returns the old value (which is either 0 or 1); and (2) *reset*, which writes 0 into r (and does not return a value).

Algorithm 2. k-OBSTRUCTION-FREE CONSENSUS FOR ANY NUMBER OF PROCESSES USING ATOMIC REGISTERS AND WAIT-FREE CONSENSUS OBJECTS FOR k PROCESSES:
`program for process` p_i `with input` in_i `(where` $in_i \in \{0,1\}$`) .`

shared registers
$x[0..\infty, 0..1]$ infinite array of bits, initially $x[0,0] = x[0,1] = 1$ and all other entries are 0
$con[1..\infty]$ infinite array of wait-free consensus objects for k processes
$decide$ ranges over $\{\bot, 0, 1\}$, initially $\bot$

local registers
r_i integer, initially 1
v_i bit, initially in_i

```
1 while $decide = \bot$ do
2     if $x[r_i, v_i] = 0$ then if $x[r_i, 1-v_i] = 1$ then $v_i := 1 - v_i$ else $x[r_i, v_i] := 1$ fi fi
3     if $x[r_i - 1, 1 - v_i] = 0$ then $decide := v_i$                // no conflict in prev round
4         else if $p_i \in set_k(r_i)$ then $v_i := con[r_i].propose(v_i)$ fi          // update pref
5     fi
6     $r_i := r_i + 1$
7 od
8 decide($decide$)
```

Fig. 7 A k-obstruction-free consensus algorithm for any number of processes.

Proof Suppose some process p commits to v in round r. Since p finds $x[r-1, 1-v] = 0$ at Statement 3, it follows that every process with preference $1-v$ for round r, will find in Statement 2 that $x[r, 1-v] = 0$ and $x[r,v] = 1$, and will change its preference to v. This implies that for a committed value v, no process ever sets $x[r, 1-v]$ to 1 in round r. It follows that all processes reaching round $r+1$ prefer v in round $r+1$, and since they will find in round $r+1$ that $x[r, 1-v] = 0$, they will all commit to v in round $r+1$. □

Lemma 12 (agreement) *No two processes decide on conflicting values.*

Proof We show that no two processes commit to conflicting values. This implies that no two processes decide on conflicting values. Assume to the contrary that two processes commit to conflicting values. This means that there exist nonfaulty processes p_0 and p_1 such that p_0 commits to 0 in round r and p_1 commits to 1 in round r'. First, suppose that $r \neq r'$. Without loss of generality, let $r < r'$. Since p_0 commits to 0 in round r, from Lemma 11 all processes reaching round $r+1$, and in particular p_1, commit to 0 in round $r+1$; a contradiction. Now suppose that $r = r'$. In round r, process p_0 commits to 0, and process p_1 commits to 1. Since process p_0 finds $x[r-1, 1] = 0$ at Statement 3, process p_1 must find $x[r, 1] = 0$ and $x[r, 0] = 1$ at Statement 2 and change it preference to 0 in round r. Consequently, it is not possible that both commit in round r. □

Lemma 13 *Let P be an arbitrary nonempty set of at most k processes and let r be a positive integer such that $P \subseteq set_k(r)$. If the processes in P complete the execution of round r and round $r+1$, before any of the other processes reach round r, then all nonfaulty processes reaching round $r+1$ (1) have the same preference, say v, for round $r+1$, and (2) commit to v either in round $r+1$ or in round $r+2$.*

Proof Assume first the processes in P all have the *same* preference v, and complete the execution of round r and round $r+1$ before any of the other process reaches round r. Since it is assumed that they execute round r without interference from the other processes, they will also reach round $r+1$ with the same preference v, and with $x[r, 1-v] = 0$. No process that will arrive later will be able to change the value of $x[r, 1-v]$. Thus, in round $r+1$ they will commit to v. Every other process that will reach round r later, will find in Statement 2 that $x[r, 1-v] = 0$ and $x[r, v] = 1$, and will change its preference to v in case it was $1-v$. This implies that no process ever sets $x[r, 1-v]$ to 1 in round r. It follows that all processes reaching round $r+1$ prefer v in round $r+1$, and since they will find in round $r+1$ that $x[r, 1-v] = 0$, they will all commit to v in round $r+1$.

Now, assume that the processes in P reach round r with *different* preferences (i.e., some prefer 0 and some 1), and complete the execution of round r and round $r+1$ before any of the other processes reach round r. Clearly, it is not possible that each one of the processes in P changes its preferences while executing Statement 2 in round r. If only the processes with input 0 (resp. input 1) change their preference while executing Statement 2 in round r, then we are in a case, similar to a one already covered, where all the processes in P reach round r with the same preference.

Thus, let us assume that not all the processes change their preference while executing Statement 2, in round r, and thus both $x[r, v]$ and $x[r, 1-v]$ will be set to 1. In such a case, Statement 4 ensures that all the processes in P will reach round $r+1$ with the same preference with the same preference v, which is the value agreed upon in $con[r]$. Since it is assumed that they execute round $r+1$ without interference from the other processes, they will also reach round $r+2$ with the same preference v, and with $x[r+1, 1-v] = 0$. No process that will arrive later will be able to change the value of $x[r+1, 1-v]$. Thus, in round $r+2$ they will commit to v. Every other process that will reach round $r+1$ later, will find in Statement 2 that $x[r+1, 1-v] = 0$ and $x[r+1, v] = 1$, and will change its preference to v in case it was $1-v$, and will later reach round $r+2$ with preference v. This implies that no process ever sets $x[r+1, 1-v]$ to 1 in round $r+1$. It follows that all processes reaching round $r+2$ prefer v, and since they will find in round $r+2$ that $x[r+1, 1-v] = 0$, they will all commit to v in round $r+2$. □

Lemma 14 (liveness with k-obstruction-freedom) *Let P be an arbitrary non-empty set of at most k processes. Each nonfaulty process eventually decides and terminates, regardless whether the other processes are faulty or not, in every run in which from some point on all the processes, except those in P, do not take any steps until the nonfaulty processes in P decide.*

Proof Assume that from some point on, say from time t, all the processes, except those in P, do not take any steps until the nonfaulty processes in P decide. Let $r_i(t)$ be the value of the register r_i at time t. Define the maximum round reached at time t, denoted $r(t)$, as: $r(t) = \text{maximum}(\{r_j(t) \mid j \in$ set of all process' identifiers$\})$. Let r' be the smallest integer such that $r(t) \leq r'$ and $P \subseteq set_k(r')$. Then, it follows from Lemma 13 that each nonfaulty process commits before or during round $r'+2$, and later executes the statement **decide**(*decide*) and decides. □

This completes the proof of Theorem 7. □

9 Extended universality results

9.1 k-universality

Lets assume that we know that o is universal in a system with k processes, what can we say about the computational power of o in a system with more than k processes? As we prove below, in such a case, o is also k-universal.

Theorem 9 (Extended Universality) *For any object o and positive integer $k \geq 1$,*

1. *A k-obstruction-free consensus object is k-universal.*
2. *o is k-universal if and only if $PN(o) \geq k$.*
3. *o is k-universal if and only if $CN(o) \geq k$.*
4. *o is k-universal if and only if o is universal for k processes.*

Proof Proving Part 1 of the theorem is difficult. So, before proving it, we first explain why all the other three statements are simple consequences of Theorem 7, Theorem 8 and Part 1 of Theorem 9. **Part 2:** If o is k-universal then, by definition, k-obstruction-free consensus can be implemented using atomic registers and objects of type o, and hence $PN(o) \geq k$. If $PN(o) \geq k$, then by Theorem 8, k-obstruction-free consensus can be implemented using atomic registers and objects of type o, and thus, by Part 1 of Theorem 9, o is k-universal. **Part 3:** This item follows immediately from Part 2 and the fact that $PN(o) = CN(o)$ (i.e., Theorem 7). **Part 4:** In a system with at most k processes, k-obstruction-free is the same as wait-freedom, thus, if an object is k-universal it is also universal for k processes. If an object, say o, is universal for k processes, by definition, o can implement consensus objects for k processes. Thus, by Theorem 8, k-obstruction-free consensus algorithm can be implemented using atomic registers and objects of type o. Thus, by Part 1 of Theorem 9, o is k-universal.

To prove Part 1 of Theorem 9, we present a universal construction that implements any k-obstruction-free object o from k-obstruction-free consensus objects and atomic registers. The construction conceptually mimics the original construction for the wait-free model from [15]. The basic idea behind the construction is as follows: an object o is implemented as a linked list which is represented as an unbounded array. The entries of the array represent a sequence of invocations applied to the object. A process invokes an operation by threading a new invocation onto the end of the list. The current state of the objects corresponds to applying the sequence of invocations to the object.

First, we need to generalize Algorithm 2. Recall that by *n-valued consensus* we mean a multi-valued consensus object where the input value taken from the set $\{0, 1, ..., n-1\}$.

Theorem 10 *For any positive integers k and n, it is possible to implement a k-obstruction-free n-valued consensus object for any number of processes, using atomic registers and k-obstruction-free binary consensus objects for any number of processes.*

Proof Starting from k-obstruction-free binary consensus objects for any number of processes, we can trivially get wait-free binary consensus objects for k processes. It is well known that using atomic registers and wait-free binary consensus objects for k processes, it is simple to implement wait-free n-valued consensus objects for k processes ([30], page 329).

In Figure 8, we show that using atomic registers and wait-free n-valued consensus objects for k processes, it is possible to implement a k-obstruction-free n-valued consensus object for any number of processes. To do that, we present below Algorithm 3 which is a simple modification of Algorithm 2.

In round $r \geq 1$, process p_i first checks if the flag of its preference v_i is already set (line 2). If it is not set and a flag for some other value is set, p_i changes its preference to the smallest such value. If non of the flags are set, p_i flags its preference v_i by writing 1 to $x[r, v_i]$ (line 4). Then, p_i reads all the flags from round $r-1$ (line 5). If non of the flags from round $r-1$ (excluding its own) is set, then every process that reaches round r with a conflicting preference will find that only $x[r, v_i]$ is set to 1, and will change its preference to v_i. Consequently, process p_i can safely decide on v_i, and it writes v_i to *decide* (line 6).

Otherwise, p_i checks if it belongs to the set $set_k(r)$. If it does, p_i proposes its current preference v_i to $con[r]$ and updates its preference to be the value agreed upon in $con[r]$ (line 7). Then, p_i proceeds to round $r + 1$. If only up to k processes with conflicting preferences participate in round r, and all of them are in $set_k(r)$, then all of them will reach round $r + 1$ with the same preference which is the value agreed upon in $con[r]$. When all processes reach a round with the same preference, a decision is reached either in that round or the next round.

The assumption that n is finite and a priori known, can be removed by replacing the while loops in lines 3 and 5, with the known snapshot algorithm for unbounded number of processes from [13]. $\square$

We assume any shared object, o, is specified by two relations:

$apply \subset$ INVOKE $\times$ STATE $\times$ STATE,

and $reply \subset$ INVOKE $\times$ STATE $\times$ RESPONSE,

where INVOKE is the object's domain of invocations, STATE is its domain of states (with a designated set of start states), and RESPONSE is its domain of responses.

1. The *apply* relation denotes a state change based on the pending invocation and the current state. Invocations do not block: it is required that for every invocation and current state there is a target state.
2. The *reply* relation determines the calculated response, based on the pending invocation and the updated state. It is required that for any pair INVOKE $\times$ STATE there is a target state and a response.

Let o be an arbitrary k-obstruction-free object which can be specified as described above. We present a universal construction that implements o from k-obstruction-free consensus objects and atomic registers. Since, by Theorem 10, k-obstruction-free n-valued consensus objects can be implemented from k-obstruction-free binary consensus objects and atomic registers, we will use in the construction below only k-obstruction-free n-valued consensus objects. The construction is similar to the one for the wait-free model from [30], where the wait-free n-valued consensus objects are replaced with k-obstruction-free n-valued consensus objects.

In the actual implementation there are two principal data structures:

1. For each process i there is an unbounded size array, *Announce*$[i][1..\infty]$, each element of which is a *cell* which can hold a single invocation. The *Announce*$[i][j]$ entry describes the j-th invocation (operation name and arguments) by process i on o.
2. The object is represented as an unbounded size array *Sequence*$[1..\infty]$ of process-id's, where for each positive integer g, *Sequence*$[g]$ is a k-obstruction-free n-valued consensus object. Intuitively, if *Sequence*$[k] = i$ and *Sequence*$[1], \ldots,$ *Sequence*$[k-1]$ contains the value i in exactly $j-1$ positions, then the k-th invocation on o is described by *Announce*$[i][j]$. In this case, we say that *Announce*$[i][j]$ has been *threaded*.

The universal construction of any k-obstruction-free object o is described in Figure 9 as the code process i executes to implement an operation on o with invocation *invoke*. For simplicity, we will assume that the input values for an *n-valued consensus* object are taken from the set $\{1, ..., n\}$ (instead of $\{0, 1, ..., n-1\}$).

In outline, the construction works as follows: process i first announces its next invocation, and then threads unthreaded, announced invocations onto the end of *Sequence*. It continues until it sees that its own operation has been threaded, computes a response, and returns. To ensure that each announced invocation is eventually threaded, the correct processes first try to thread any announced, unthreaded cell of process ℓ into entry *Sequence*$[k]$, where $\ell = k \pmod{n} + 1$. This "helping" technique guarantees that once process ℓ announces an operation, at most n other operations can be threaded before the operation of process ℓ is threaded.

Process i keeps track of the first index of *Announce*$[i]$ that is vacant in a variable denoted *MyNextAnnounce*, and first writes the invocation into *Announce*$[i]$[*MyNextAnnounce*], and (line 2) increments *MyNextAnnounce* by 1. To keep track of which cells it has seen threaded (including its own), process i keeps n counters in an array *NextAnnounce*$[1..n]$, where each *NextAnnounce*$[j]$ is one plus the number of times i has read cells of j in *Sequence*. Hence *NextAnnounce*$[j]$ is the index of *Announce*$[j]$ where i looks to find the next operation announced by j. We notice that, having incremented *MyNextAnnounce*:

NextAnnounce$[i] =$ *MyNextAnnounce* $- 1$ until the current operation of process i has been threaded.

Algorithm 3. k-OBSTRUCTION-FREE n-VALUED CONSENSUS FOR ANY NUMBER OF PROCESSES USING ATOMIC REGISTERS AND WAIT-FREE n-VALUED CONSENSUS OBJECTS FOR k PROCESSES:
`program for process` p_i `with input` in_i `(where` $in_i \in \{0, 1, ..., n-1\}$`)` .

shared registers
$x[0..\infty, 0..n-1]$ infinite array of bits, initially entries of $x[0..n-1]$ are 1, all other entries are 0
$con[1..\infty]$ infinite array of wait-free n-valued consensus objects for k processes
$decide$ ranges over $\{\perp, 0, 1, ..., n-1\}$, initially $\perp$

local registers
r_i integer, initially 1 ; v_i integer, initially in_i ; j integer

```
1  while $decide = \perp$ do
2      if $x[r_i, v_i] = 0$ then
3          $j := 0$; while $j < n$ and $x[r_i, j] = 0$ do $j := j + 1$ od
4          if $j < n$ then $v_i := j$ else $x[r_i, v_i] := 1$ fi fi
5      $j := 0$; while($v_i = j$) or ($j < n$ and $x[r_i - 1, j] = 0$) do $j := j + 1$ od
6      if $j = n$ then $decide := v_i$                          // no conflict in previous round
7          else if $p_i \in set_k(r_i)$ then $v_i := con[r_i].propose(v_i)$ fi      // update pref
8      fi
9      $r_i := r_i + 1$
10 od
11 decide($decide$)
```

Fig. 8 A k-obstruction-free n-valued consensus algorithm for any number of processes.

This inequality is thus the condition (line 3) in the while loop (lines 3 – 13) in which process i threads cells. Once process i's invocation is threaded (and $NextAnnounce[i] = MyNextAnnounce$), it exits the loop and returns the associated response value (line 14). Process i keeps an index *NextSeq* which points to the next entry in *Sequence*$[1..\infty]$ whose element it has not yet accessed.

To thread cells, process i proposes (line 9) the id of process ℓ to the k-obstruction-free consensus object *Sequence*[*NextSeq*], and after a decision is made, records the consensus value for *Sequence*[*NextSeq*] in the local variable *Winner* (line 9). The value in *Sequence*[*NextSeq*] is the identity of the process whose cell has just been threaded. After choosing to help process ℓ (line 4), process i checks that *Announce*$[\ell]$[*NextAnnounce*$[\ell]$] contains a valid operation invocation. As discussed above, process i gives preference (line 4) to a different process for each cell in *Sequence*. Thus, all active processes will eventually agree to give preference to any pending invocation, ensuring it will eventually be threaded.

Once process i knows the id of the process whose cell has just been threaded, as recorded in *Winner*, it can update (line 10) its view of the object's state with the winner invocation, and increment its records of process *Winner*'s successfully threaded cells (line 11) and the next unread cell in *Sequence* (line 12). Having successfully threaded a cell, process i returns to the top of the while loop (line 3). Eventually, the invocation of process i will be threaded and the condition at the while loop (line 3) will be *false*. At this point, the value of the variable *CurrentState* is the state of the object after process i's invocation has been applied to the object. Based on this state, process i can return the appropriate response. This completes the proof of Theorem 9. □

9.2 S-universality

We generalize the notion of wait-free universality using the notion of S-freedom (from Section 3). An object o is *S-universal* for n processes if any object which has sequential specification has an S-free linearizable implementation using registers and objects of type o for n processes.

One of the important results proved in [15], is that wait-free consensus for n processes is universal for n processes. Below we generalize this result.

Theorem 11 *For any positive integer n, and any non-empty set $S \subseteq \{1, ..., n\}$, an S-free consensus object for n processes is S-universal for n processes.*

To prove the result, we need to present, as done in Section 9.1, a universal construction that implements any S-free object o for n processes from S-free consensus objects for S processes and registers. This is done by taking the universal construction from Section 9.1 (i.e., Algorithm 4) and replacing the infinite array *Sequence*$[1..\infty]$ of k-obstruction-free

ALGORITHM 4. A UNIVERSAL CONSTRUCTION:
`program for process` $i \in \{1,...,n\}$ `with invocation` *invoke*

```
shared
  Announce[1..n][1..∞] array of cells which range over INVOKE ∪ {⊥},
      initially all cells are set to ⊥
  Sequence[1..∞] array of k-obstruction-free n-valued consensus objects
local to process i
  MyNextAnnounce integer, initially 1                          // next vacant cell
  NextAnnounce[1..n] array of integers, initially 1
                                                               // next operation
  CurrentState ∈ STATE, initially the initial state of o       // i's view
  NextSeq integer, initially 1                                 // next entry in Sequence
  Winner range over {1,...,n}                                  // last process threaded
  ℓ range over {1,...,n}                                       // process to help
                                         // write invoke to a vacant cell in Announce[i]
1   Announce[i][MyNextAnnounce] := the invocation invoke
2   MyNextAnnounce := MyNextAnnounce + 1
3   while ((NextAnnounce[i] < MyNextAnnounce) do
                                               // continue until invoke is threaded
                                               // each iteration threads one operation
4       ℓ := NextSeq (mod n) + 1                                // select process to help
5       while Announce[ℓ][NextAnnounce[ℓ]] = ⊥                  // valid?
7       do
6           ℓ := ℓ + 1                                          // not valid; help next process
7       od
9       Winner := Sequence[NextSeq].propose(ℓ)                  // propose ℓ
                                               // a new cell has been threaded by Winner
                                                                // update CurrentState
10      CurrentState := apply(Announce[Winner][NextAnnounce[Winner]], CurrentState)
11      NextAnnounce[Winner] := NextAnnounce[Winner] + 1
12      NextSeq := NextSeq + 1
13  od
14  return(reply(invoke, CurrentState))
```

Fig. 9 A Universal Construction.

n-valued consensus objects with an infinite array (with the same name) of S-free n-valued consensus objects. This is the only change needed; the explanation is the same as that in Section 9.1.

We point out that by Lemma 8, S-free n-valued consensus objects can be implemented from S-free binary consensus objects and registers. For simplicity, we will assume that the input values for an *n-valued consensus* object are taken from the set $\{1, ..., n\}$ (instead of $\{0, 1, ..., n-1\}$). This completes the proof of Theorem 11. An immediate corollary of Theorem 11 is,

Corollary 1 *For any object o, any positive integer n, and any non-empty set $S \subseteq \{1, ..., n\}$, o is S-universal for n processes if and only if an S-free consensus object for n processes can be implemented from objects of type o and registers.*

10 Extending the definitions of progress conditions

It is possible to extend the definitions of progress conditions in various ways. Below we define two such new interesting extensions.

- For any non-empty set $S \subseteq \{1, ..., n\}$ and an integer $1 \leq k \leq n$, the progress condition (S, k)*-freedom* guarantees that for every set of processes P, if at some point in a computation $active.P = |P|$ and $|P| \in S$, then (at least) $\min\{k, |P|\}$ processes in P will be able to eventually complete their pending operations, provided that (1) all the processes not in P do not take steps for long enough; and (2) none of the processes in P fails.
- Let $W_1, ..., W_n$ be n sets of sets of process identifiers such that $P \in W_i$ only if $p_i \in P$. The progress condition $(W_1, ..., W_n)$*-freedom* guarantees that for every set

of processes P and every process p_i, if at some point in a computation $active.P = |P|$ and $P \in W_i$, then process p_i will be able to eventually complete its pending operations, provided that (1) all the processes not in P do not take steps for long enough; and (2) none of the processes in P fails.

We notice that in a system of n processes,

- (S, n)-*freedom* is the same as S-*freedom*;
- $(\{1, ..., n\}, 1)$-freedom is the same as a known condition called *non-blocking* [19] (sometimes also called lock-freedom);
- $(\{1, ..., n\}, k)$-freedom is the same as *k-lock-freedom* [4]; several other special cases of (S, n)-*freedom* were considered recently in [4];
- Adversaries are a way to express progress conditions [5]. Each one of the *adversaries* considered in [5] corresponds to some $(W_1, ..., W_n)$-free progress condition, which has the following property: For every set P, if $P \in W_i$ and $p_j \in P$ then $P \in W_j$. See the related work section for more explanation regarding adversaries.

11 Related work

The consensus problem was formally defined in [27]. The impossibility result that there is no consensus algorithm that can tolerate even a single crash failure in an asynchronous model was first proved for the message-passing model in [8], and later has been extended for the shared memory model in which only atomic registers are supported in [25]. A recent survey which covers many related impossibility results can be found in [6].

The power of shared objects has been studied extensively in environments where processes may fail benignly, and where every operation is wait-free. In [15], Herlihy classified objects by their consensus numbers and defined the wait-free hierarchy. Additional results regarding the wait-free hierarchy can be found in [21,23].

Objects that can be used, together with registers, to build wait-free implementations of any other object are called *universal objects*. Previous work provided methods, called universal constructions, to transform sequential specifications of arbitrary shared objects into wait-free concurrent implementations that use universal objects [15,28]. In [28] it is proved that sticky bits are universal, and independently, in [15] it is proved that wait-free consensus objects are universal. A bounded space version of the universal construction from [15] appears in [22]. The universal construction that we use to prove Theorem 9 conceptually mimics the original construction from [15] and is similar to the one for the wait-free model from [30]. Linearizability is defined in [19].

Two extensively studied progress conditions are wait-freedom [15] and obstruction-freedom [16]. It is shown in [16] that obstruction-free consensus is solvable using registers. Various contention management techniques have been proposed to improve obstruction-freedom under contention [10,29]. Other works investigated boosting obstruction-freedom by making timing assumption [3,7] and using failure detectors [12]. Wait-free consensus algorithms that use registers in the absence of contention and revert to using strong synchronization operations when contention occurs are presented in [2,24,26].

The notion of *asymmetric* progress conditions was coined in [20], where the (n, x)-liveness condition which guarantees wait-freedom for x processes and obstruction-freedom for the remaining $n - x$ processes, was defined. The following results are proven in [20]: (1) It is not possible to implement an $(n, 1)$-*live* consensus object using wait-free consensus objects for $n - 1$ processes and registers; (2) For $1 \leq x < n - 1$, an (n, x)-live consensus object is strictly weaker than an $(n, x + 1)$-live consensus object, thereby establishing a hierarchy for (n, x)-liveness; (3) It is not possible to implement a consensus object for n processes which guarantees both fault-freedom and obstruction-freedom for one process and only obstruction-freedom for the remaining $n - 1$ processes, using wait-free consensus objects for $n - 1$ processes and registers; (4) It is possible to implement a consensus object for $n \geq x$ processes that satisfies a condition called asymmetric group-based progress condition using (x, x)-live consensus objects and registers. Asymmetric progress conditions are related to the notion of an adversarial scheduler studied in the context of non-uniform failure patterns in [17].

The notion of k-obstruction-freedom is presented in [31, 32], as part of a transformation that is used to fuse objects which avoid locking and locks together in order to create new types of shared objects. Similar notions were suggested independently in [9]. A weaker set of progress conditions, called k-obstacle-freedom, which cover the spectrum between obstruction-freedom and non-blocking (which is sometimes called lock-freedom) is defined in [31,32]. The non-blocking progress condition was first defined in [19].

In [18], the authors identify an interesting relationship that unifies six progress conditions ranging from the deadlock-free and starvation-free conditions common to lock-based systems, to the obstruction-free, non-blocking and wait-free conditions common to lock-free systems. The authors also proposed a new condition, called clash-freedom, which is weaker than obstruction-freedom. These six conditions are classified along two dimensions: (1) those that ensure maximal progress, that is, progress for all processes, from conditions that ensure minimal progress, progress for only some process, and (2) those that depend on different kinds of guarantees provided by the operating system scheduler.

Adversaries were extensively studied in [5,11]. Intuitively, an adversary can choose which set of processes will crash.

More precisely, an *adversary* defined for a system of processes P is a non-empty set of process subsets $A \subseteq 2^P$. The system is required to make progress only in executions in which the set of non-correct (faulty) processes is in A. Thus, an adversary controls sets of processes that may fail in a given execution, regardless of the time when they fail [5]. Although progress conditions and adversaries are two seemingly different notions, they are actually closely related, and should be viewed as two sides of the same coin. Progress conditions can be expressed as adversaries, and vice versa.

Progress conditions and adversaries are simply two different formalisms for expressing which executions are *legitimate* executions, that is, the executions in which the correct processes are required to make progress and eventually produce correct results. For example, the classical n process t-resilient adversary, denoted R^n_t, is the adversary for which at most t $(0 \leq t \leq n-1)$ processes may crash: $R^n_t = \{u \subseteq \{p_1, ..., p_n\} |\ |u| \leq t\}$. Thus, the adversary R^n_t corresponds to the $\{n-t, ..., n\}$-freedom progress condition, defined in Section 3. Similarly, the adversary $O^n_k = \{u \subseteq \{p_1, ..., p_n\} |\ |u| \geq n-k\}$ corresponds to the $\{1, ..., k\}$-freedom progress condition, called k-*obstruction-freedom*, defined in Section 3.

In [5], a precise way is presented to characterize adversaries by introducing the notion of disagreement power: the biggest integer k for which the adversary can prevent processes from agreeing on k values when using registers only; and it is shown how to compute the disagreement power of an adversary. As already mentioned in Section 10, each one of the *adversaries* considered in [5] corresponds to some $(W_1, ..., W_n)$-free progress condition, which has the following property: For every set P, if $P \in W_i$ and $p_j \in P$ then $P \in W_j$.

Solvability in the presence of various adversaries is further explored in [11]. In particular, the authors of [11] have determined necessary and sufficient conditions for an adversary A to solve consensus for n processes using wait-free consensus objects for $k < n$ processes and read-write registers. It is possible to easily derive from the results of [11], alternative proofs for Theorem 1 and Theorem 6.

12 Discussion

Several interesting research directions are: exploring the complexity and computability of problems like set-consensus, renaming, etc. under various new progress conditions; exploring the relationship to failure detectors, by possibly extending known results for wait-freedom [14]; better understanding of the relations between different progress conditions; adding timing assumptions. Known open problems, like the robustness of the wait-free hierarchy or whether a queue object can be implemented from a set of test-and-set objects, fetch-and-add objects, swap objects and atomic registers, for $n \geq 3$, can now be studied in our more general setting.

The study should not be limited to shared memory systems only. Consider for example n senders that are trying to broadcast the same message to a single receiver, and it is required that at least one of the senders succeeds to transmit, without collisions, whenever an *odd* number of senders broadcast at the same time. This required progress condition, and similar ones, that are sometimes expressed using the notion of a conflict graph, can be easily formally expressed and studied within our general framework.

We conclude with few additional interesting open problems. With respect to which set of objects, denoted O, the set of progress conditions proposed in this paper is strict? Namely, can we find two different sets S_1 and S_2 such that S_1-freedom is equivalent to S_2-freedom, assuming that only objects from O can be used? In the paper, linearizability is used as the only consistency condition. What happens if we assume sequential consistency or weak consistency condition like causal consistency? Can we refine the power hierarchy using the set-consensus problem, as was recently done for the wait-free hierarchy in [1]?

Acknowledgments: I wish to thank the editor Eric Ruppert and three anonymous referees for their constructive comments and suggestions.

References

1. Y. Afek, F. Ellen, and E. Gafni. Deterministic Objects: Life Beyond Consensus. *In Proceedings of the 35th ACM Symp. on Principles of Distributed Computing*, pages 97-106, 2016.
2. H. Attiya, R. Guerraoui, and P. Kouznetsov. Computing with reads and writes in the absence of step contention. *Proceedings of the 19th International Symposium on Distributed Computing*, LNCS 3724, 122–136, 2005.
3. M. K. Aguilera and S. Toueg. Timeliness-based wait-freedom: a gracefully degrading progress condition. In *Proc. 27rd ACM Symp. on Principles of Distributed Computing*, pages 305–314, 2008.
4. V. Bushkov and R. Guerraoui. Safety-Liveness Exclusion in Distributed Computing. In *Proc. 2015 ACM Symposium on Principles of Distributed Computing*, pages 227-236, 2015.
5. C. Delporte-Gallet, H. Fauconnier, R. Guerraoui, and A. Tielmann. The disagreement power of an adversary. *Distributed Computing*, 24(3):137–147, 2011.
6. F. E. Fich and E. Ruppert. Hundreds of impossibility results for distributed computing. *Distributed Computing*, 16(2-3):121–163, 2003.
7. E. F. Fich, V. Luchangco, M. Moir, and N. Shavit. Obstruction-free algorithms can be practically wait-free. *Proc. of the 19th International Symp. on Distributed Computing*, LNCS 3724, pp. 78-92, 2005.
8. M.J. Fischer, N.A. Lynch, and M.S. Paterson. Impossibility of distributed consensus with one faulty process. *Journal of the ACM*, 32(2):374–382, 1985.
9. E. Gafni and R. Guerraoui. Simulating Few by Many: Limited Concurrency = Set Consensus. *Unpublished manuscript*, 2009.

10. R. Guerraoui, M. P. Herlihy and B. Pochon. Towards a theory of transactional contention managers. In *Proc. of the 24th Symp. on Principles of Dist. Computing*, pp. 258–264, 2005.
11. E. Gafni and P. Kuznetsov. Turning Adversaries into Friends: Simplified, Made Constructive, and Extended. In *14th International Conference on Principles of Distributed Systems (OPODIS 2010). LNCS 6490* Springer Verlag 2010, 380–394.
12. R. Guerraoui, M. Kapalka and P. Kouznetsov. The weakest failure detectors to boost obstruction-freedom. *Distributed Computing*, 20(6):415–433, 2008.
13. E. Gafni, M. Merritt, and G. Taubenfeld. The concurrency hierarchy, and algorithms for unbounded concurrency. In *Proc. 20th ACM Symp. on Principles of Distributed Computing*, pages 161–169, August 2001.
14. R. Guerraoui and P. Kouznetsov. Failure detectors as type boosters. *Distributed Computing*, 20:343–358, 2008.
15. M. P. Herlihy. Wait-free synchronization. *ACM Trans. on Programming Languages and Systems*, 13(1):124–149, January 1991.
16. M. P. Herlihy, V. Luchangco, and M. Moir. Obstruction-free synchronization: Double-ended queues as an example. In *Proc. of the 23rd Int. Conf. on Dist. Computing Systems*, 2003.
17. M. Herlihy and S. Rajsbaum. The topology of distributed adversaries. *Distributed Computing*, 26(3):173–192, 2013.
18. M. Herlihy and N. Shavit. On the nature of progress. In *15th International Conference on Principles of Distributed Systems (OPODIS 2011)*, 2011. *LNCS 7109* Springer Verlag 2011, 313-328.
19. M. P. Herlihy and J. M. Wing. Linearizability: a correctness condition for concurrent objects. *TOPLAS*, 12(3):463–492, 1990.
20. D. Imbs, M. Raynal, and G. Taubenfeld. On asymmetric progress conditions. In *Proc. 29th ACM Symp. on Principles of Distributed Computing*, pages 55–64, 2010.
21. P. Jayanti. Robust wait-free hierarchies. *Journal of the ACM*, 44(4):592–614, 1997.
22. P. Jayanti and S. Toueg. Some results on the impossibility, universality, and decidability of consensus. In *Proc. of the 6th Int. Workshop on Distributed Algorithms: LNCS 674*, pages 69–84, 1992.
23. Wai-Kau Lo and Vassos Hadzilacos. All of us are smarter than any of us: Nondeterministic wait-free hierarchies are not robust. *SIAM Journal on Computing*, 30(3):689–728, 2000.
24. V. Luchangco, M. Moir and N. Shavit. On the uncontended complexity of consensus. *Proc. of the 17th International Symp. on Distributed Computing*, LNCS 2848, 45–59, 2003.
25. M.C. Loui and H. Abu-Amara. Memory requirements for agreement among unreliable asynchronous processes. *Advances in Computing Research*, 4:163–183, 1987.
26. M. Merritt and G. Taubenfeld. Resilient consensus for infinitely many processes. *Proc. of the 17th International Symp. on Distributed Computing*, LNCS 2848, 1–15, 2003.
27. M. Pease, R. Shostak, and L. Lamport. Reaching agreement in the presence of faults. *Journal of the ACM*, 27(2):228–234, 1980.
28. S. A. Plotkin. Sticky bits and universality of consensus. In *Proc. 8th ACM Symp. on Principles of Distributed Computing*, pages 159–175, 1989.
29. W. N. Scherer and M. L. Scott. Advanced contention management for dynamic software transactional memory. In *Proc. of the 24th Symp. on Principles of Dist. Computing*, pp. 240-248, 2005.
30. G. Taubenfeld. Synchronization Algorithms and Concurrent Programming. *Pearson / Prentice-Hall*, ISBN 0-131-97259-6, 423 pages, 2006.
31. G. Taubenfeld. Contention-sensitive data structures and algorithms. *Proc. of the 23rd International Symp. on Distributed Computing*, Spain, 2009. LNCS 5805 , 157–171, 2009.
32. G. Taubenfeld. Contention-sensitive data structures and algorithms. *Theoretical Computer Science*, 677:41–55, May 2017.
33. G. Taubenfeld. On the computational power of shared objects. *Proc. of the 13th international conf. on principles of distributed systems*, France, 2009. LNCS 5923, 270–284, 2009.
34. G. Taubenfeld. The computational structure of progress conditions. *Proc. of the 24th International Symp. on Distributed Computing*, USA, 2010. LNCS 6343, 221–235, 2010