The Computational Structure of Progress Conditions

(FULL VERSION – SEPTEMBER 10, 2010)

Gadi Taubenfeld

The Interdisciplinary Center, P.O.Box 167, Herzliya 46150, Israel,
tgadi@idc.ac.il,
http://www.faculty.idc.ac.il/gadi/

Abstract. Understanding the effect of different progress conditions on the computability of distributed systems is an important and exciting research direction. For a system with n processes, we define exponentially many new progress conditions and explore their properties and strength. We cover all the known, symmetric and asymmetric, progress conditions and many new interesting conditions. Together with our technical results, the new definitions provide a deeper understanding of synchronization and concurrency.

Keywords: Progress conditions, wait-freedom, obstruction-freedom, S-freedom, consensus, synchronization, contention, cooperation, universality, hierarchy.

1 Introduction

We define exponentially many new progress conditions and explore their properties and relative strength. Our results regarding the computational structure of the new and known, symmetric and asymmetric, progress conditions provide a deeper understanding of synchronization and concurrency. Most of the known progress conditions can be classified as either cooperation-based conditions or contention-based conditions. Cooperation arises when several processes need to coordinate their actions in order to achieve a common goal. Contention arises when several processes compete for exclusive use of shared resources, such as communication bandwidth, data items or files.

Fault-freedom, the weakest cooperation-based condition, guarantees that every process will complete its pending operations provided that all the processes participate and there are no failures. *Obstruction-freedom*, the weakest contention-based condition, guarantees that a process will be able to complete its pending operations in a finite number of its own steps, if all the other processes "hold still" (i.e., do not take any steps) long enough [11]. *Wait-freedom*, the strongest both contention-based and cooperation-based progress condition, guarantees that every process will always be able to complete its pending operations in a finite number of its own steps, regardless of the behavior of the other processes [10]. While a consensus object can be implemented using only atomic registers under either fault-freedom or obstruction-freedom, it can not be implemented using registers under wait-freedom.

We start by proving two general impossibility results for symmetric progress conditions, which have many interesting implications. For example, we show that objects that satisfy cooperation-based progress conditions can be implemented from objects that satisfy the corresponding contention-based conditions, but not vice versa. We establish a formal connection between symmetric and asymmetric progress conditions, which enables us to apply the general impossibility results for proving new results also for asymmetric progress conditions. For the special case where only atomic registers are used, we give a complete characterization under which symmetric progress conditions consensus is solvable, and prove impossibility results for the asymmetric case. Finally, we prove a general universality result.

1.1 Exponentially many symmetric progress conditions

From now on we assume that the number of processes is n and $n \geq 2$. A process is *active* when it has pending operations, otherwise it is *passive*. 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 active processes in P. We use S to denote a non-empty set such that $S \subseteq \{1, ..., n\}$.

Definition. For any non-empty set S, the progress condition S*-freedom* guarantees that for every set of processes P, if at some point in a computation $active.P = |P|$ and $|P| \in S$, then every process in P 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 (which means that each of the processes in P will continue to take steps until it becomes passive).

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. Since not all the processes in P become passive at the same time eventually some process in P will become passive and once this happens $active.P = k - 1$. It is important to notice, that although at this point $active.P = k-1 \notin S$, the definition of *S-freedom* guarantees that (because in the near past all k processes were active) eventually every process in P will become passive.

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

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 the same as fault-freedom; $\{1\}$-freedom is the same as obstruction-freedom; $\{1, ..., n\}$-freedom is the same as wait-freedom; $\{1, n\}$-freedom is the progress condition which implies both obstruction-freedom and fault-freedom. For $1 \leq k \leq n$, $\{1, ..., k\}$-freedom is the same as k-obstruction-freedom. We call these n conditions, *contention-based* progress conditions, since $\{1, ..., k\}$-freedom guarantees progress under contention of at most k processes. For $0 \leq t \leq n-1$, $\{n-t, ..., n\}$-freedom is the same as t-resiliency. We call these n conditions, *cooperation-based* progress conditions, since $\{n-t, ..., n\}$-freedom captures the ability to tolerate t faults.

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 conditions in the sense that a given process is not favored with respect to the others.

1.2 Asymmetric progress conditions

The notion of *asymmetric* progress conditions was coined and investigated in [13]. It is motivated by the observation that some processes may be more important than others and hence should get stronger progress guarantees. Thus, an asymmetric progress condition specifics the progress guarantees for each process separately. One such progress condition which is defined in [13], called (n, x)-liveness, satisfies wait-freedom for x processes and satisfies obstruction-freedom for the remaining $n - x$ processes.

In the literature, saying that an *object* is wait-free is the same as saying that each one of the *processes* is wait-free w.r.t. that object. Although using the term wait-freedom in two different ways may be confusing, it simplifies the discussion. Following this "tradition", we will say that an object is S-free iff each process is S-free w.r.t. that object. Clearly, a process that is T-free is also S-free, for any $S \subset T$. Asymmetric progress conditions can be practically motivated by modern multicore architectures where processes in different cores might be provided with different progress guarantees.

1.3 Our contributions

A consensus object o supports one operation: $o.propose(v)$ satisfying: (1) *Agreement*: In any run, the $o.propose()$ operation 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 $v \in \{0, 1\}$ the object is called a binary consensus object. By a consensus object we mean a binary consensus object; and by *n-consensus* we mean a multi-valued consensus object where $v \in \{0, 1, ..., n - 1\}$. The term *register* means an atomic read/write register. A simple (but not obvious) observation is that, for two positive integers m and n, and a set $S \subseteq \{1, ..., \min(m, n)\}$, it is *not* always possible to implement an S-free consensus object for n processes using S-free consensus objects for m processes and registers. Our results are:

New Definitions. We define exponentially many progress conditions and investigate their properties and relative strength. Together with the technical results, the new notions provide a deeper understanding of synchronization and concurrency.

General impossibility results. Let S be a subset of $\{1, ..., n\}$, where $n \geq 2$. $|S|$ 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: $width.S = 1 + max.S - min.S$. We prove the following two impossibility results,

- For any set $|S| \geq 2$, 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.

– 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 not possible 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 the results that: (1) for any $2 \leq k \leq n$, it is not possible to implement a $\{1, k\}$-free consensus object for k processes using any number of wait-free consensus objects for $n - 1$ processes and registers; and (2) For any $n > 2$, it is not possible to implement a wait-free consensus object for two processes using any number of $\{1, n\}$-free consensus objects for n processes and registers.

Cooperation vs. contention. It follows from the above impossibility results that objects which satisfy cooperation-based progress conditions can not be used to implement objects which satisfy contention-based progress conditions. However, objects that satisfy cooperation-based conditions can be implemented from objects that satisfy the corresponding contention-based conditions. More formally,

– It is not possible to implement $\{1, 2\}$-free consensus object for n processes using $\{2, ..., n\}$-free consensus objects for n processes and registers. However, 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.

This result is rather surprising, given the fact that while cooperation-based conditions imply fault-freedom, contention-based conditions do not imply the fault-freedom.

Asymmetric progress conditions. We establish a connection between symmetric and asymmetric conditions, which enables us to apply the general impossibility results for proving new results also for asymmetric conditions. For example, we show that:

– For any two integers k_1 and k_2 such that $1 \leq k_1 < k_2 \leq n$, it is not possible to implement a consensus object for n processes that satisfies $\{k_1, k_2\}$-freedom for $n - k_2 + 1$ processes and satisfies $\{k_1\}$-freedom for all the other processes, using any number of wait-free consensus objects for $k_2 - k_1$ processes and registers.

Atomic registers. When only registers are used, we have a complete characterization under which symmetric conditions consensus is solvable, and prove impossibility results for the asymmetric case. For the symmetric case, we show that:

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

The results generalize the famous FLP result for the case of one faulty process [6, 18].

Universality. We generalize results regarding the universality of consensus from [10]. 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. We prove that,

– 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.

The result implies that an object 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. The *wait-free hierarchy* [10], is an infinite hierarchy of objects, such that the objects at level i are exactly those objects which are $\{1, ..., i\}$-universal for i processes, but are not $\{1, ..., i+1\}$-universal for $i+1$ processes. We will explain, how to define other interesting hierarchies.

1.4 Related work

The consensus problem was formally defined in [20]. 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 [6], and later has been extended for the shared memory model in which only atomic registers are supported, in [18]. A recent survey which covers many related impossibility results can be found in [4]. The power of shared objects has been studied extensively in environments where processes may fail benignly, and where every operation is wait-free. In [10], Herlihy classified objects by their consensus numbers and defined the wait-free hierarchy. Additional results regarding the wait-free hierarchy can be found in [14, 16].

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 [10, 21]. In [21] it is proved that sticky bits are universal, and independently, in [10] it is proved that wait-free consensus objects are universal. A bounded space version of the universal construction from [10] appears in [15]. Linearizability is defined in [12].

Two extensively studied conditions are wait-freedom [10] and obstruction-freedom [11]. It is shown in [11] that obstruction-free consensus is solvable using registers. Various contention management techniques have been proposed to improve obstruction-freedom under contention [7, 22]. Other works investigated boosting obstruction-freedom by making timing assumption [1, 5] and using failure detectors [8]. 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, 17, 19].

The notion of *asymmetric* progress conditions was coined in [13], 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 [13]: (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.

The notion of k-obstruction-freedom is presented in [24], 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. In [25], a new classification for evaluating the strength of shared objects is proposed. The classification is based on finding, for each object of type o, the largest k for which it is possible to solve consensus for any number processes, using any number of objects of type o and registers, assuming that the required progress condition is k-obstruction-freedom. The main technical result in [25] is that the new classification is equivalent to Herlihy's traditional classification.

Although progress conditions and adversaries are two seemingly different notions, they are actually closely related. In [3], 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. Our formalism for expressing progress conditions is not expressive enough to express all the adversaries considered in [3], and vice versa. In the last section, we generalize our formalism to express both.

2 Preliminaries

Our model of computation consists of an asynchronous collection of n processes that communicate via shared objects. 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 does 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 implementation of an object from a set of other objects, consists of a non-empty set C of runs, a set N of processes, and a set of shared objects O. 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 $(f,\ R;T)$.

Process p is *enabled* at run x if there exists an event e_p such that $x;e_p$ is a run. For simplicity, we write xp to denote either $x;e_p$ when p is enabled in x, or x when p is not 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. For example, 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. In the following proofs, we will implicitly make use of few such straightforward properties.

3 Impossibility results

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 $width.S \geq |S|$.

Theorem 1. *For any set $|S| \geq 2$, 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.*

It follows immediately from Theorem 1 that for any $2 \leq k \leq n$, it is not possible 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 point out that it follows from a result in [25] that, when $1 \in |S|$, it is possible to implement an S-free consensus object for n processes using wait-free consensus objects for $width.S$ (which in this case equals $max.S$) processes and registers. Next we consider the relative strength of different 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 not possible 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 not possible to implement a wait-free consensus object for two processes using any number of $\{1, n\}$-free consensus objects for n processes and registers. Next we prove the theorems.

3.1 A detailed proof

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 (i.e. $x[N - P]y$), 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 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 $p \in Q$ decide v (i.e., p writes v to its output register). Clearly $z = y; (x - w)$ is also a run of the algorithm such that $z[Q]x$. Since p writes v to its output register in z, z is (P, v)-valent. Hence, since $y \leq z$, y must also be (P, v)-valent. □

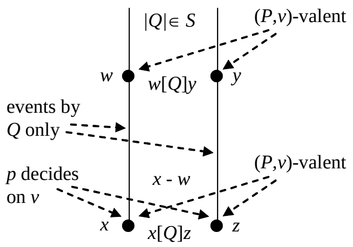

Fig. 1. Illustration of runs in the proof of Lemma 2.

Lemma 3. *Let $|S| \geq 2$ and let P be a set of processes such that $|P| = max.S$. Then, every 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 $|Q| = min.S$. By Lemma 2, 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.

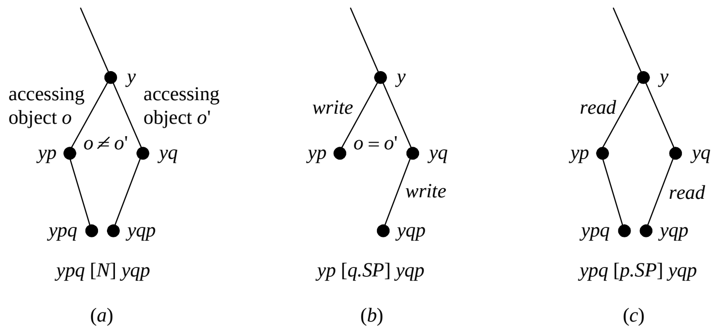

Fig. 2. Illustration of runs in the proof of Lemma 4.

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. $\square$

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 $x[\overline{P}]y$ /* involves $p_i$ */
4         then $x := yp_i$              /* $P$-bivalent extension of $x$ */
5         else $Q := p_i$          /* no such $P$-bivalent extension */
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. (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.*

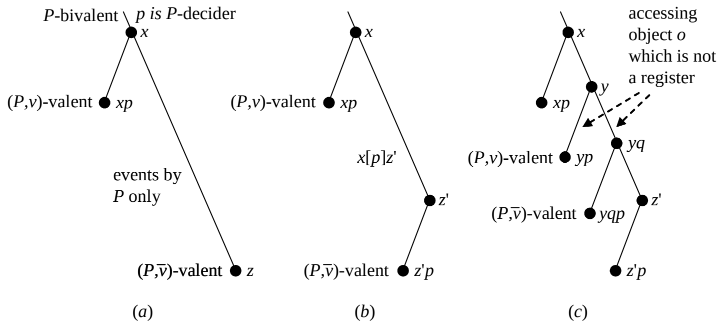

Fig. 3. Illustration of runs in the proof of Lemma 6.

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).)

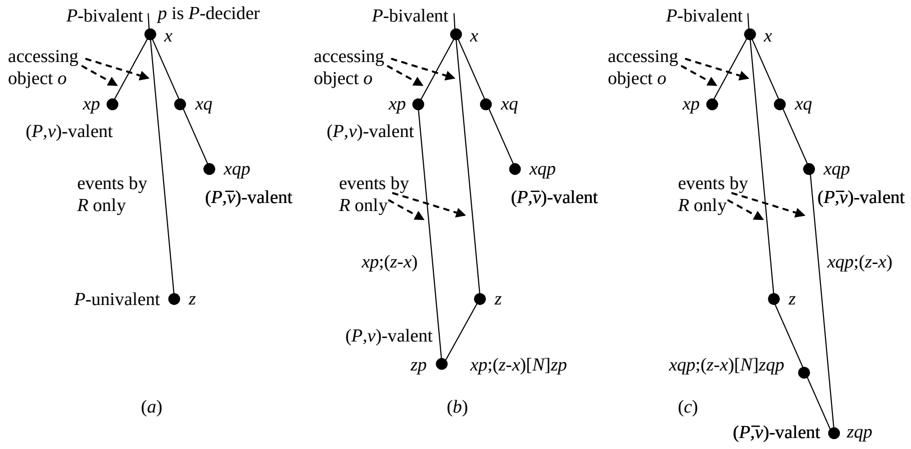

Fig. 4. Illustration of runs in the proof of Lemma 7.

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* 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. $\square$

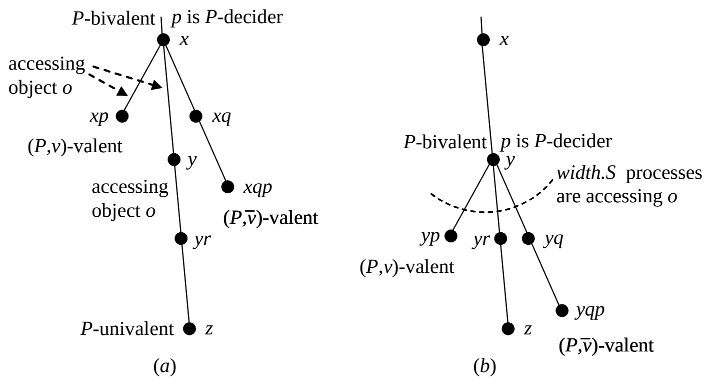

Fig. 5. Additional illustration of runs in the proof of Lemma 7.

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. $\square$

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.

Assume that at the end of y, just before the $width.T$ processes access o, $n - k$ processes fail and the remaining k active processes, including p and q, are about to access o. Since there are only k active processes and $k \in T$, the implementation of a T-free consensus object must guarantee that these k 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. Assume none of the k processes ever gets a response for o. Although the k processes may continue to take steps, because yp and yqp are P-univalent and not P-compatible, the final decision value (of the T-free consensus object) depends on getting a response from o. Without a response from o, it is not possible to determine whether the prefix of the current run is yp or yqp. Thus, the k processes will never be able to terminate. A contradiction. $\square$

4 Cooperation vs. contention

It follows from the impossibility results that objects which satisfy cooperation-based progress conditions can not implement objects which satisfy contention-based progress conditions. More formally,

Theorem 3. *It is not possible 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 objects that satisfy cooperation-based conditions can be implemented from objects that satisfy the corresponding contention-based conditions.

Theorem 4. *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 4, we first generalize a known result for wait-freedom, namely, that multi-valued consensus can be implemented from binary consensus ([23], page 329).

Lemma 8. *For any $k \geq 2$, $n \geq 2$ and $S \subseteq \{1, ..., n\}$, an S-free k-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-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. □

Proof of Theorem 4: 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-consensus object. Each participating process is progressing from a leaf to the root, where at each level of the tree it accesses a k-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 left most subtree, of process $k - 1$ when it arrives from the right most 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. $\square$

5 Asymmetric progress conditions

As already mentioned, the notion of asymmetric progress conditions was coined and investigated in [13]. Let *APC* be an Asymmetric Progress Condition; we define *max.APC*, *min.APC* and *width.APC* as follows,

- *max.APC* is the largest $1 \leq k \leq n$ such that (at least) $n - k + 1$ processes are $\{k\}$-free, or 0 if no such k exists.
- *min.APC* is the smallest $1 \leq k \leq n$ such that *every* process is $\{k\}$-free, or 0 if no such k exists.
- $width.APC$ equals $1 + max.APC - min.APC$ if $min.APC \neq 0$, or 0 otherwise.

Thus, for the asymmetric progress condition $(n, 1)$-liveness (as defined in [13]), $max.(n, 1)$-$liveness = n$, $min.(n, 1)$-$liveness = 1$ and $width.(n, 1)$-$liveness = n$.

Lemma 9. *Let O be a consensus object for n processes that satisfies an asymmetric progress condition APC such that $min.APC \geq 1$. Using O and a single register it is possible to implement a consensus object for n processes that satisfies the symmetric progress condition (min.APC, max.APC)-freedom.*

Proof. Let *decision* be a register which is initially set to -1. Each process tries to reach a decision by accessing O. A process that reaches a decision writes the decision value into *decision* and terminates. Each process infinitely often reads *decision*, and if the value read is different from -1, it decides on that value and terminates. Since every subset of *max.APC* processes includes at least one $\{max.APC\}$-free process, this implementation clearly satisfies (*min.APC*, *max.APC*)-freedom. Another way to view this implementation is: once O returns a value to some processes, it keeps this value in an internal private register, and thereafter returns it immediately to every process that accesses it. $\square$

Lemma 10. *Let APC be a an asymmetric progress condition such that $1 \leq min.APC < max.APC \leq n$, and let P be a set of processes such that $|P| = max.APC$. Every consensus object for n processes that satisfies APC has a P-bivalent run y, a set $Q \subseteq P$ of size $width.APC$, 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 $width.APC$ processes in Q, are accessing the same object, and this object is not a register.*

Proof. Assume to the contrary that O is a consensus object for n processes that satisfies *APC*, and O does not have a run y with *all* the three properties as mentioned in Lemma 10. By Lemma 9, using O and a single register it is possible to implement a consensus objects O' for n processes that satisfies the symmetric progress condition (*min.APC*, *max.APC*)-freedom. Thus, also O' does not have such a a run y. However, this contradicts Lemma 7. $\square$

Theorem 5. *For any asymmetric progress condition APC such that $1 \leq min.APC < max.APC \leq n$, it is not possible to implement a consensus object for n processes that satisfies APC using any number of wait-free consensus objects for $width.APC - 1$ processes and registers.*

Proof. The proof is similar to that of Theorem 1. It follows from Lemma 10 that every implementation of a consensus object for n processes that satisfies *APC*, must use an object, say o, which at least $width.APC$ processes must be able to access at the same run, and o is not a register. Thus, it is not possible to implement a consensus object for n processes that satisfies APC using any number of wait-free consensus objects for $width.APC - 1$ processes and registers. □

It is proven in [13] that it is not possible to implement an $(n, 1)$-*live* consensus object using any number of wait-free consensus objects for $n-1$ processes and registers; and that this result holds even when the requirement that one process should be wait-free is replaced with the much weaker requirement that one process is $\{1, n\}$-free. These important results are special cases of the following corollary of Theorem 5.

Corollary 1. *For any two positive integers k_1 and k_2 such that $1 \leq k_1 < k_2 \leq n$, it is not possible to implement a consensus object for n processes, that satisfies $\{k_1, k_2\}$-freedom for $n - k_2 + 1$ processes and satisfies $\{k_1\}$-freedom for all the other processes, using any number of wait-free consensus objects for $k_2 - k_1$ processes and registers.*

Another interesting result from [13] is that: 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. Using Lemma 10 it is possible to slightly generalize this result.

6 Atomic registers

For the case where only registers are used, we present a complete characterization under which symmetric progress conditions consensus is solvable, and prove impossibility results for the asymmetric case.

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 $|S| = 1$.*
- *For any asymmetric progress condition APC, it is not possible to implement a consensus object for n processes that satisfies APC using registers if $width.APC > 1$.*

Proof. It follows from Theorem 1 that it is not possible to implement an S-free consensus object for n processes using registers if $|S| \geq 2$, and it follows from Theorem 5 that it is not possible to implement a consensus object for n processes that satisfies APC using registers if $width.APC > 1$.

Next, 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 (i.e., implementation) 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.

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.

IMPLEMENTING $\{k\}$-FREE CONSENSUS FOR n PROCESSES USING REGISTERS (WHERE $k \in \{1,...,n\}$): `program for process` p_i `with input` in_i `(where` $in_i \in \{0,1\}$ `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
$\mathit{flag}[1..\infty, 1..n]$ 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($decide$)
```

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 $\mathit{flag}[r_i, i]$ (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).

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 Universality

In [10], the notion of universality is introduced in the context of wait-freedom. An object o is (wait-free) *universal* for n processes if any object which has sequential specification has a wait-free linearizable implementation using registers and objects of type o in a system with n processes. Below we generalize the notion of wait-free universality.

> **Definition.** 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 [10], is that wait-free consensus for n processes is universal for n processes. Next we generalize this result.

Theorem 7. *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 present a universal construction that implements any S-free object o for n processes from S-free consensus objects for S processes and registers. The construction conceptually mimics the original construction for the wait-free model from [10]. In Subsection 7.1 below, we give such a construction which is similar to the one for the wait-free model from [23]. A similar type of a universality result (with a similar proof) can be proved also for asymmetric progress conditions.

Corollary 2. *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.*

The *wait-free hierarchy* is an infinite hierarchy of objects, introduced in [10], such that the objects at level i of the hierarchy are exactly those objects which are $\{1, ..., i\}$-universal for i processes, but are not $\{1, ..., i + 1\}$-universal for $i + 1$ processes. For that hierarchy, by the above definition, (1) no object at level less than i together with registers can implement any object at level i; and (2) each object at level i together with registers can implement any object at level i or at a lower level.

The wait-free hierarchy is meaningful because it can be defined using only the (contention-based) 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. Similar such hierarchies, in which there is a total order between the conditions, can be naturally defined. For example, by using the cooperation-based progress conditions, the cooperation hierarchy can be defined as follows: For a given system of n processes, the objects at level i of the hierarchy are exactly those objects which are $\{n-i+1, ..., n\}$-universal for n processes, but are not $\{n - i, ..., n\}$-universal for n processes.

7.1 A universal construction

To prove Theorem 7, we present below a universal construction that implements any S-free object o for n processes from S-free consensus objects for n processes and registers. 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. We assume any shared object, o, is specified by two relations:

$$apply \subset \text{INVOKE} \times \text{STATE} \times \text{STATE},$$

$$\text{and } reply \subset \text{INVOKE} \times \text{STATE} \times \text{RESPONSE},$$

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 an arbitrary S-free object which can be specified as described above. We present a universal construction that implements o from S-free consensus objects and registers.

In the actual implementation there are two principal data structures:

1. For each process i there is an unbounded 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 array *Sequence*$[1..\infty]$ of process-id's, where for each positive integer k, *Sequence*$[k]$ is a S-free n-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 S–free object o is described below as the code process i executes to implement an operation on o with invocation *invoke*. Since, by Lemma 8, S-free n-consensus objects can be implemented from S-free binary consensus objects and registers, we will use in the construction below only S-free n-consensus objects. For simplicity, we will assume that the input values for an *n-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.

A UNIVERSAL CONSTRUCTION:
`program for process` $i \in \{1,\ldots,n\}$ `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 S-free n-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))
```

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.

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 S-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 7. $\square$

8 Discussion

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

Definition. 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.

We notice that in a system of n processes, (S, n)-*freedom* is the same as S-*freedom*; and $(\{1, ..., n\}, 1)$-freedom is the same as a known condition called *non-blocking* [12] (sometimes also called lock-freedom).

Definition. Let $W_1, ..., W_n$ be sets of sets of process identifiers such $P \in W_i$ 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.

Each one of the adversaries considered in [3] 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 $P \in W_j$. We notice that satisfying this property, completely precludes the ability to express the asymmetric progress conditions defined in the introduction. That is, w.r.t. this definition, this property distinguishes between symmetric and asymmetric progress conditions (adversaries).

Additional interesting questions are: exploring the complexity and computability of problems like set-consensus, renaming, etc. under various new progress conditions; exploring the relation to failure detectors, by possibly extending known results for wait-freedom [9]; defining meaningful hierarchies; 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.

References

1. 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.
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. C. Delporte-Gallet, H. Fauconnier, R. Guerraoui, and A. Tielmann. The disagreement power of an adversary. *Proc. of the 23rd International Symp. on Distributed Computing*, Spain, September 2009. LNCS 5805 , 8–21, 2009.
4. F. E. Fich and E. Ruppert. Hundreds of impossibility results for distributed computing. *Distributed Computing*, 16(2-3):121–163, 2003.
5. 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.
6. 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.
7. 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.
8. R. Guerraoui, M. Kapalka and P. Kouznetsov. The weakest failure detectors to boost obstruction-freedom. *Distributed Computing*, 20(6):415–433, 2008.
9. R. Guerraoui and P. Kouznetsov. Failure detectors as type boosters. *Distributed Computing*, 20:343–358, 2008.
10. M. P. Herlihy. Wait-free synchronization. *ACM Trans. on Programming Languages and Systems*, 13(1):124–149, January 1991.

11. 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.
12. M. P. Herlihy and J. M. Wing. Linearizability: a correctness condition for concurrent objects. *TOPLAS*, 12(3):463–492, 1990.
13. 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.
14. P. Jayanti. Robust wait-free hierarchies. *Journal of the ACM*, 44(4):592–614, 1997.
15. 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.
16. 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.
17. 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.
18. M.C. Loui and H. Abu-Amara. Memory requirements for agreement among unreliable asynchronous processes. *Advances in Computing Research*, 4:163–183, 1987.
19. 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.
20. M. Pease, R. Shostak, and L. Lamport. Reaching agreement in the presence of faults. *Journal of the ACM*, 27(2):228–234, 1980.
21. S. A. Plotkin. Sticky bits and universality of consensus. In *Proc. 8th ACM Symp. on Principles of Distributed Computing*, pages 159–175, 1989.
22. 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.
23. G. Taubenfeld. Synchronization Algorithms and Concurrent Programming. *Pearson / Prentice-Hall*, ISBN 0-131-97259-6, 423 pages, 2006.
24. G. Taubenfeld. Contention-sensitive data structures and algorithms. *Proc. of the 23rd International Symp. on Distributed Computing*, Spain, 2009. LNCS 5805 , 157–171, 2009.
25. 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.