Distributed Computing (2022) 35:375–398
https://doi.org/10.1007/s00446-022-00425-x

Extending the wait-free hierarchy to multi-threaded systems

Matthieu Perrin[1] · **Achour Mostéfaoui**[1] · **Grégoire Bonin**[1] · **Ludmila Courtillat-Piazza**[2]

Received: 29 October 2020 / Accepted: 19 March 2022 / Published online: 16 April 2022

Abstract
In modern operating systems and programming languages adapted to multicore computer architectures, parallelism is abstracted by the notion of *execution threads*. Multi-threaded systems have two major specificities: on the one part, new threads can be created dynamically at runtime, so there is no bound on the number of threads participating in long-running executions. On the other part, threads have access to a memory allocation mechanism that cannot allocate infinite arrays. These specificities make it challenging to adapt some algorithms to multi-threaded systems, in particular those that need to assign one shared register per process. This paper explores the synchronization power of shared objects in multi-threaded systems by extending the famous Herlihy's wait-free hierarchy to take these constraints into consideration. It proposes to subdivide the set of objects with an infinite consensus number into nine new degrees, depending on their ability to synchronize a bounded, finite or infinite number of processes, with or without the need to allocate an infinite array. To show the relevance of the proposed extension, for each new degree it is either proved that it is empty, or an object illustrating it is proposed.

Keywords Arrival models · Consensus number · Memory allocation · Multi-threaded system · Universality · Wait-freedom

1 Introduction

1.1 Wait-free universality

In sequential computing, the notion of universality is represented by a Turing machine capable of computing anything that is computable. Read/write registers, the basic objects of a Turing machine, are thus universal objects in sequential computing. In the context of distributed systems, we know, since 1985 and the famous FLP impossibility result, that the consensus problem has no deterministic solution in a distributed system where even one process might fail by crashing [10]. This impossibility is not due to the computing power of the individual processes, but rather to the difficulty of coordination between the different processes that compose the distributed system. Coordination and agreement problems are thus at the heart of computability in distributed systems [13].

A shared memory distributed system can be abstracted as a set of processes accessing concurrently a set of shared objects. The implementations of these objects are based on read/write registers and hardware instructions. Searching for correct and efficient implementations of usual objects (e.g. queues, stacks) is far from being trivial when the system is failure prone [14,19,21]. Intuitively, a "good" implementation of a concurrent object has to satisfy two properties: a consistency condition and a progress condition that specify respectively the meaningfulness of the returned results, and the guarantees on the liveness.

Linearizability [15] is a consistency condition. It ensures that all the operations of a distributed history appear as if they were executed sequentially: each operation on an object appearing at a single point in time, between its start and end events. This gives the illusion to the processes to access a physical concurrent object.

This work was partially supported by the French ANR project 16-CE25-0005 O'Browser (https://obrowser.univ-nantes.fr/) devoted to the study of decentralized applications on Web browsers.

✉ Matthieu Perrin
matthieu.perrin@univ-nantes.fr

Achour Mostéfaoui
achour.mostefaoui@univ-nantes.fr

Grégoire Bonin
gregoire.bonin@univ-nantes.fr

Ludmila Courtillat-Piazza
ludmila.courtillat-piazza@ens-rennes.fr

[1] LS2N, Université de Nantes, Nantes, France

[2] École Normale Supérieure de Rennes, Bruz, France

The use of locks in an implementation may cause blocking in a system where processes can crash. Prohibiting the use of locks leads to several progress conditions, among which wait-freedom [12] and lock-freedom [15]. While wait-freedom guarantees that every operation terminates after a finite time, lock-freedom guarantees that, if a computation runs for long enough, at least one process makes progress (this may lead some other processes to starve). Wait-freedom is thus stronger than lock-freedom: while lock-freedom is a system-wide progress condition, wait-freedom is a per-process progress condition.

A major difficulty of distributed computing is that wait-free linearizable implementations are often costly, when not impossible. In the latter, the system has to be enriched with some more sophisticated objects or hardware special instructions. The coordination power of objects is thus important for computability in distributed systems. In [12], consensus is proved universal. Namely, any object having a sequential specification has a wait-free implementation using only read/write registers and some number of consensus objects.

Hence the idea to assign to each object a consensus number representing its ability to solve consensus. More precisely, an object has consensus number x if it is universal in an asynchronous system composed of x processes, but not in a system composed of $x + 1$ processes. If no upper bound exists on x, the object has an infinite consensus number.

1.2 Problem statement

This last decade, first with peer-to-peer systems, and then with multi-threaded programs on multicore machines, the assumption of a closed system with a fixed number n of processes and where every process knows the identifiers of all processes became too restrictive. In multi-threaded systems, new processes can be created and started at run-time, so although the number of processes at each time instant is finite, there is no bound on the total number of processes that can participate in long-running executions.

Another specificity of multi-threaded systems must be taken into account. Threads share a (virtually) unbounded common memory space. As in the Java and C languages, the processes have access to a primitive to allocate this memory (`new` or `malloc`). Such calls should specify the number of memory locations they ask for. By this mean and during its execution time each thread can allocate an unbounded but finite number of memory locations. This memory allows to instantiate record data structures or arrays.

It turns out that many synchronization algorithms require the sharing of an array whose size depends on the number of processes to be synchronized (e.g. The bakery mutual exclusion algorithm [16]). This may be problematic, when no bound is known on the number of threads in an execution: assigning one register to each of them is not trivial, especially if this number can be infinite. This fact is often regarded as secondary when designing concurrent algorithms. For example, [4] identifies as "trivial" the change of finite arrays indexed by processes to infinite arrays or linked lists. Among other contributions of this paper, the fact that maintaining extensible data structures such as linked lists requires synchronization power that is not necessarily provided by all objects which have an infinite consensus number.

The two aspects noted above have an important impact on which algorithms can be implemented in multi-threaded systems and which algorithms cannot, and therefore on the coordination power of shared objects: in [2], Afek, Morrison and Wertheim exhibited an object called the *iterator stack* (noted `IStack`) that has an infinite consensus number, but cannot be used to implement consensus when infinitely many processes may join an execution over time. The present paper answers the following question: how to compare the synchronization power of shared objects in multi-threaded systems?

1.3 Approach

Following the same approach as in [12], we propose to compare the synchronization power of shared objects based on the maximal number of processes they are able to synchronize, including in situations where the set of participating processes is initially unknown or may change during an execution. More precisely, we differentiate computing models according to the restrictions on process arrival, as introduced in [11]. In these models, any number of processes may crash (or leave, in a same way as in the classical model), but fresh processes can also join the network during an execution. When a process joins such a system, it is not known to the already running processes. Three (families of) arrival models are distinguished in [4]:[1]

– For each integer $n \geq 1$, the n-arrival model M_1^n, where the number n of processes is fixed and may appear in the process code. As stated in [11], "work on adaptive algorithms implicitly precludes the use of the system size n as a parameter in a solution." Hence, we generalize

[1] A fourth model, M_4, called *infinite concurrency*, was introduced in [4], where infinitely many processes may be present in the system and an infinite number of operations can take place in any finite interval of time. We choose to ignore this model because it poses a problem to define linearizability.
Suppose that, for each $i \geq 1$, process p_i writes the value i in a variable x during the interval $\left[1 - \frac{1}{2i}; 1 - \frac{1}{2i+1}\right]$; then p_0 starts reading x at time 1. There is no "last written value" before the read, so the return value is not well defined. Restricting infinite concurrency to a subset of non-conflicting operations (*e.g.* reads or operations on different objects) would render infinite concurrency and infinite arrival computationally equivalent as one can easily use contention on conflicting operations to control the arrival of processes.

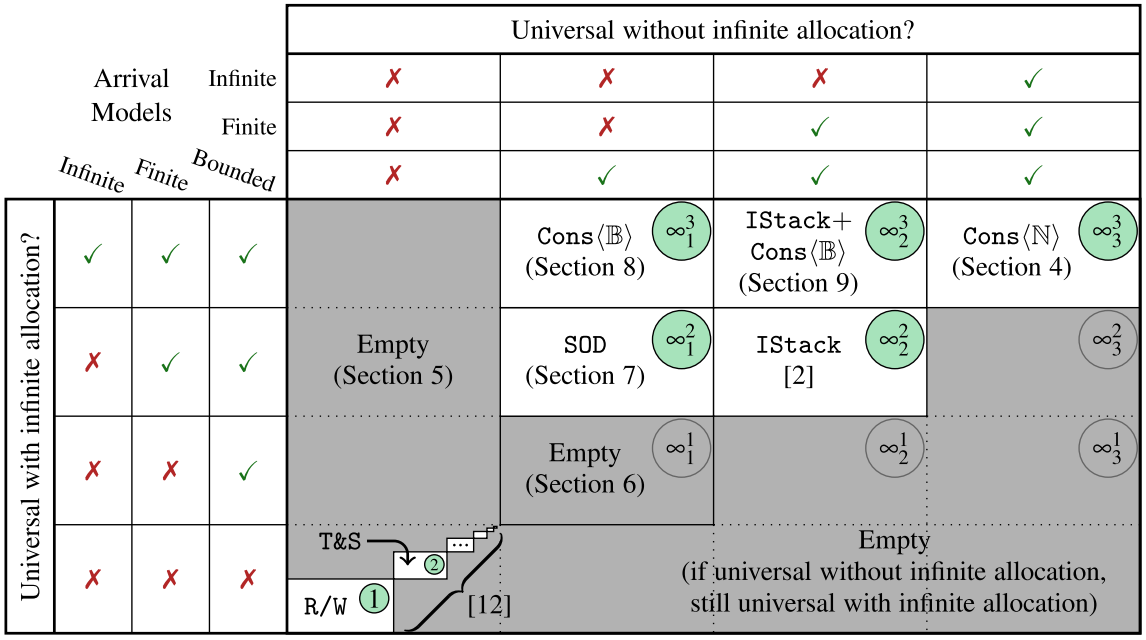


Fig. 1 Extended wait-free hierarchy: in a multi-treaded system, it is impossible to implement an object O_1 using any number of instances of O_2 and read/write registers, if O_2 is more on the left, or bottom, than O_1. Green circles display the consensus number of each degree

the model family into the bounded arrival model, M_1, in which at most n processes may participate, where n is only known, to the processes, at the beginning of each execution, but may vary from one execution to another.
- The finite arrival model, M_2, in which a finite number of processes participate in each execution.
- The infinite arrival model, M_3 (also referred to as unbounded concurrency model), where new processes may keep arriving during the whole execution. Let us note that, at any time, the number of processes that have already joined the system is finite, but can be infinitely growing.

We pointed out above that the impossibility to allocate an array with an infinite range of indices is a major limiting factor that restricts the computing power of some objects in multi-threaded systems. We thus study the power of synchronization of shared objects depending on whether the possibility of allocating infinite arrays is offered or not. Therefore, we propose the two-dimensional hierarchy presented on Fig. 1. In this hierarchy, shared objects are sorted horizontally depending on their universality in models M_1^n, M_1, M_2 and M_3 when infinite memory allocation is not available, and vertically on their ability to do so when it is possible to allocate infinite arrays. We then challenge the significance of this hierarchy by exploring whether or not there exists an object filling each possible degree.

1.4 Contributions of the paper

In a first step, we show how the proposed hierarchy encompasses the existing one and then for each new degree, either a representative object is proposed or it is proved empty.

Extend the wait-free hierarchy We show that, on the one hand, the proposed hierarchy coincides with Herlihy's hierarchy on objects with a finite consensus number. Indeed, Theorem 2 proves that infinite arrays are not necessary for universal constructions in models M_1^n and M_1, which justifies that we keep the same term "consensus number" to categorize shared objects in our hierarchy. On the other hand, the proposed hierarchy refines the one proposed by Herlihy for objects with infinite consensus number. We say that an object O has consensus number ∞_x^y, for $x, y \in \{1, 2, 3\}$ if O is universal in M_x but not M_{x+1} (if $x \neq 3$) when infinite memory allocation is not available, and O is universal in M_y but not M_{y+1} (if $y \neq 3$) when infinite memory allocation is available. As having access to infinite arrays is never detrimental, no object has consensus number ∞_x^y for $y < x$.

Identify all filled degrees Following our approach, we prove that no object has consensus number ∞_1^1 (Theorem 3), and we identify objects filling all remaining degrees of the hierarchy, as depicted by Fig. 1. We prove that multi-valued consensus (denoted $\texttt{Cons}\langle\mathbb{N}\rangle$) is still universal in all the

models considered in this paper, i.e. it has consensus number ∞_3^3. Rephrasing the theorems concerning the iterator stack [2], we naturally deduce that iterator stacks have consensus number ∞_2^2. Interestingly, we prove that binary consensus (denoted $\texttt{Cons}\langle\mathbb{B}\rangle$) is not universal in multi-threaded systems, resulting in a consensus number of ∞_1^3 (Theorem 5). The proof that the composition of binary consensus and iterator stacks has consensus number ∞_2^3 (Theorem 6) is the most technical part of the paper. We define a new special instruction, called `setOrDecrement` and denoted `SOD`, that either writes its parameter or decrements the register depending on the sign of its previous value, and we show that a register provided with the `setOrDecrement` operation has consensus number ∞_1^2 (Theorem 4).

1.5 Organization of the paper

The remainder of this paper is organized as follows. We first illustrate the practical issues that led to our problem statement in Sect. 2 and, then, we present the infinite arrival models in Sect. 3. Section 4 shows that consensus is still universal in the infinite arrival model. Sections 5 and 6 identify the empty degrees of the hierarchy by proving theorems 2 and 3. Sections 8, 7, and 9 show that the remaining degrees are not empty by proving the consensus number of `setOrDecrement` registers, binary consensus and a composition of binary consensus and iterator stacks. Finally, Sect. 10 concludes the paper.

2 Illustration of the issue

Sharded counters Several implementations of shared counters are available for concurrent programs. One may simply use the `fetchAndAdd` instruction, if available. Another solution is to protect the incrementation of a single shared variable using a `compareAndSwap` instruction, within a loop that retries until the `compareAndSwap` is successful.[2] Such a strategy has the drawback of creating contention on the single variable, that may impact performance when many processes try to access the variable simultaneously. Another possibility, similar to the *sharded counter* in cloud computing, consists in assigning to each process p a single-writer/multi-reader register, which only p can safely increment without fear of concurrent updates. A read of the shared counter is then the sum of all the values obtained after a snapshot of the set of registers. This strategy is at the basis of the `LongAdder` class in the standard library of Java, but this class is neither wait-free nor linearizable. In the following, we will discuss how we can implement a sharded counter in Java.

The bounded arrival model The simplest implementation of a sharded counter uses an array of size n, where n is the number of threads. However, n must be known by the constructor when creating the array. Passing n as an argument to the constructor might not be a problem for some programs. For example, when parallelism is managed through a fixed-size thread pool, the size of the thread pool can be used for the size of the array. Programs for which a limit on the number of threads created throughout an execution is a priori known are said to belong to the *bounded arrival model*, denoted by M_1.

The finite arrival model Now imagine a program that must first create a shared counter, then read a configuration file, starting a new thread for each selected option. Here, the assumption of a fixed n known at the start of execution may not be realistic. One solution could be to allocate arrays that are much larger than the size that will be used in practice. This approach has several drawbacks. Indeed, in the case where the limit of the allocated array is effectively reached, the correctness and the termination of the algorithms are no longer ensured. The rest of the time, this approach presents a huge waste. It would therefore be more judicious to have dynamic data structures having the possibility of arbitrarily growing to adapt to the number of threads, such as dynamic arrays, linked lists, sets, or dictionaries.

Simple linked lists can be designed following the same algorithmic pattern as the push operation of Treiber's stack [22]. The constructor of such an algorithm only creates a sentinel cell (the end of the linked list) as well as a shared register, head (reference to this sentinel cell). The first time a thread accesses the shared object, it allocates its own cell and, in an infinite loop, attempts to insert it at the head of the list, using a `compareAndSwap`.

An important question is the following: is the algorithm described above wait-free? Since there is an infinite loop, it is possible that some process always loses its `compareAndSwap`, and never terminates its operation. However, this cannot happen in the example of the configuration file described above, because no new thread is created after the configuration file has been completely parsed, and each process only needs to win the `compareAndSwap` once. The hypothesis that there is an instant after which no new thread is started is called the *finite arrival model*, denoted by M_2.

The infinite arrival model Finally, let us consider a server that starts a new thread each time it receives a request from a client. If the server is properly sized, the number of threads running at any time may remain relatively low. Yet, the linked list algorithm described above is only lock-free, as there are executions in which some thread may never be able to insert its own cell. The most general model, in which there is no

[2] This was the standard implementation of `getAndIncrement` in Java, prior to version 8.

assumption about the number of threads that can be created during an execution, is called the *infinite arrival model* and denoted by M_3.

The example of the sharded counter above illustrates a common issue when trying to adapt many distributed algorithms to multi-threaded programming languages such as Java or C++: how to deal with data structures whose size depends on the number of participants? This paper explores one facet of this issue: the synchronization power necessary, and sufficient, to build any wait-free data structure.

3 Computing models

This paper considers distributed computations where processes (or threads) have access to local memory for local computations and also have access to shared objects (shared memory) to communicate and synchronize with each other. We define, below, the assumptions on the set of processes and the kind of memory they can access. Each combination of a process model and a memory model instantiates a different computing model. Moreover, as some objects cannot be implemented using only read/write registers, a system can be enriched with synchronization objects like consensus objects, iterator stacks, etc, or by special instructions that can be invoked on registers, such as `setOrDecrement`.

3.1 Arrival models

We consider computation models composed of a set Π of sequential processes $p_0, p_1, \ldots$ Each process p_i has a unique identifier i that may appear in its code. The set Π is the set of potential processes that may join, get started and crash or leave during a given execution. At any time, the number of processes that have joined is finite. The cardinality of Π defines four computing models:

- n-arrival models M_1^n ($n \geq 1$): $|\Pi| = n$, and n is a parameter of each system model.
- Bounded arrival model M_1: Π is finite and $|\Pi|$ is known to the processes at runtime. In other words, M_1 is the union of the M_1^n, for all n, and a problem can be solved in the bounded arrival model if, and only if, it can be solved in the n-arrival model, regardless the size of the system.
- Finite arrival model M_2: Π is finite but $|\Pi|$ is unknown to the processes.
- Infinite arrival model M_3: Π is countable.

3.2 Communication between processes

Processes communicate by reading and writing a memory composed of an infinite number of unbounded registers[3]. Reads and writes on a shared register x are denoted by x.`read`() and x.`write`(v). We also consider local variables and read-only shared registers, for which we use the lighter notations x and $x \leftarrow v$.

Processes have access to a dynamic allocation mechanism that can only return an unbounded, but finite, number of memory locations at once. The allocation mechanism is accessible through the syntax `new` T, that instantiates an object of type T (T may be a record datatype or a shared object giving access to a set of operations) and returns its reference, i.e. it allocates the memory locations needed to manage the object and initializes them by calling a constructor.[4]

Processes are not limited in the number of registers they can access, nor by the number of times they can use the allocation mechanism, during an execution. However, they can only access memory locations that either 1) have been allocated at the system set up, or 2) they obtained directly through the allocation mechanism, or 3) are accessible by following references stored (as integer values) in some accessible memory location. In other words, when a process p_i allocates a memory location at runtime, it can initially only be accessed by p_i until p_i shares a reference pointing to it with other processes.

As advocated in the Introduction, when sufficiently powerful synchronization objects are not available, it may be necessary to assume an allocation mechanism which allows to allocate and initialize an infinite number of memory locations at once. When a system allows such allocation, it is said to provide infinite allocation. This defines four more arrival system models MA_1^n, MA_1, MA_2 and MA_3 that represent the four above-mentioned models enriched with an infinite memory allocation mechanism. In our algorithms, infinite arrays are accessible through a type `InfiniteArray`, whose constructor takes, as a parameter, a rule $i \mapsto f(i)$ stating that the cell at index i must be set to $f(i)$. Like in finite arrays, the cell at index i of an array A is denoted by $A[i]$.

3.3 Synchronization objects

In order to improve their computability, the different computing models can be enriched by giving access to more evolved shared atomic objects, that are denoted between

[3] Memory addresses of an infinite memory are unbounded, so this assumption is necessary to store references.

[4] In this paper, we do not consider a de-allocation or garbage collection mechanism, because we only investigate computability issues that are not affected by the possibility to reuse memory locations.

square brackets in the model name and referred to as enriching shared objects. For example, $M_3[\texttt{Cons}\langle\mathbb{N}\rangle]$ denotes the infinite arrival model where as many consensus objects as necessary are made available.

Set-or-decrement registers A set-or-decrement register `SOD` is an integer register providing the standard `read` and `write` operations, as well as a `setOrDecrement` special instruction that takes an integer as argument and has no return value. An invocation of `setOrDecrement`(v) first reads the current value x of the register. If $x \leq 0$, the register is set to v. Otherwise, the register is decremented by one.

Iterator stacks The iterator stack `IStack`, introduced in [2], provides a write operation `isWrite`() and a read operation `isRead`(). Intuitively, `isWrite`(v) prepends the value v at the beginning of a stack and returns a reference i to a fresh iterator, and `isRead`(i) increments iterator i and returns the value it points to. More precisely, `isWrite`(v) takes a written value $v \in \mathbb{N}$ as argument and returns the next integer value in a sequence $0, 1, \ldots$ For a given $i \in \mathbb{N}$, the kth invocation of `isRead`(i) returns the kth value ever written if `isWrite` was invoked at least $\max(i + 1, k)$ times, and $\perp$ otherwise.

Consensus objects A consensus object, denoted $\texttt{Cons}\langle T\rangle$, provides two operations. The operation `propose`(v) takes an argument $v \in T$ and returns the oldest proposed value, i.e. the first process that invokes the operation gets its own value and all invocations returns this same value, called decision value and we say that the consensus object is won. A second operation, `get`(), returns the value stored in the consensus object if it has been won; otherwise, it returns a default value $\perp$. We distinguish between binary consensus $\texttt{Cons}\langle\mathbb{B}\rangle$ in which only two values can be proposed (e.g. **true** and **false**), and the multi-valued consensus, for example $\texttt{Cons}\langle\mathbb{N}\rangle$ in which proposed values are integer values, possibly encoding a reference to a memory location. Finally, $\texttt{Cons}^n\langle\mathbb{N}\rangle$ designates the n-process consensus that only has a `propose`(v) operation that only verify the previously-stated properties for its first n invocations, and the next returned values are left unspecified.

3.4 Distributed executions

An execution α is a (finite or infinite) sequence of steps, each taken by a process of Π. A step of a process corresponds to the execution of a hardware instruction or an operation of one of the atomic enriching objects defined above. Processes are asynchronous, in the sense that there is no constraint on which process takes each step: a process may take an unbounded number of consecutive steps, or wait an unbounded but finite number of other processes' steps between two of its own steps. Moreover, it is possible that a process stops taking steps at some point in the execution, in which case we say this process has *crashed*, or even that a process takes no step during a whole execution ($|\Pi|$ is only an upper bound on the number of participating processes). We say that a process p_i *arrives* in an execution at the time of its first step during this execution. Remark that, although the number of processes in an execution may be infinite in M_3, the number of processes that have arrived into the system at any step is finite.

A configuration C is composed of the local state of each process in Π and the internal state of each enriching shared object, including read/write registers. For a finite execution α, we denote by $C(\alpha)$ the configuration obtained at the end of α. An empty execution is noted ε. An execution β is an extension of α if α is a prefix of β.

Implementation of shared objects An implementation of a shared object is an algorithm divided into a set of sub-algorithms, one for the initialization (a.k.a. the constructor of the object), and one for each operation of the object, that produces wait-free and linearizable executions. Linearizability and atomicity are equivalent thanks to observational refinement, i.e. if an object O has a linearizable implementation in a model M, then M and $M[O]$ are computationally equivalent ($M[O]$ represents the model M enriched with atomic objects O) [9].

Definition 1 (*Linearizability*) An execution α is *linearizable* if all operations return the same value as if they occurred instantly at some point of the timeline, called the *linearization point*, between their invocation and their response, possibly after removing some non-terminated operations.

Definition 2 (*Wait-freedom*) An execution α is *wait-free* if no operation takes an infinite number of steps in α.

Consensus protocols Similarly to [12], it may be useful to express consensus as a one-shot task [10], i.e. one in which each process proposes some value and may decide a value, such that the three following properties are respected:

- Wait-freedom (see Definition 2).
- Validity: all decided values are proposed by some process.
- Agreement: distinct processes never decide on distinct values.

Previous affirmations that one-shot consensus and wait-free linearizable consensus objects are computationally equivalent [18] still apply in all models used in this papers, so we use both definitions interchangeably.

Universality A model M is said to be *wait-free universal* (or simply universal) if any object with a sequential specification can be implemented in M, with respect to linearizability and wait-freedom. By extension, an object O is said to be universal in M if $M[O]$ is universal.

Let O be an object. We say that O has consensus number $n \in \mathbb{N}$ if $M_1^n[O]$ is universal but not $M_1^{n+1}[O]$, and that O

has consensus number ∞_x^y, for $x, y \in \{1, 2, 3\}$ if it verifies both following conditions:

- $M_x[O]$ is universal and, if $x \leq 2$, then $M_{x+1}[O]$ is not universal.
- $MA_y[O]$ is universal and, if $y \leq 2$, then $MA_{y+1}[O]$ is not universal.

Remark that the proposed hierarchy is not strict: it is impossible to use any number of objects with consensus number ∞_1^3 to implement an object with consensus number ∞_2^2 in a multi-threaded system, because this would require allocating infinite arrays. Conversely, it is impossible to implement an object with consensus number ∞_1^3 using only objects with consensus number ∞_2^2 because some participating processes could starve while new processes constantly arrive in the system.

4 Universality of consensus in M_3

The aim of this section is to extend universality of consensus to multi-threaded systems. In order to prove the universality of consensus in the bounded arrival model, Herlihy introduced the notion of universal construction[5]. It is a generic algorithm that, given a sequential specification of any object whose operations are deterministic and total,[6] provides a concurrent implementation of this object. Wait-free implementations rely on what is called a helping mechanism, recently formalized in [6]. This mechanism requires that, before terminating its operation, a process helps completing pending ones of other processes. Helping is not obvious in the infinite arrival model. Indeed, a process should be able to announce itself to processes willing to help it. However, due to the infinite number of potential participating processes over time, it is not reasonable to assume that each process can write in a dedicated register that can be read by all.

Similarly to [8] which first proposes a Collect object and then uses it as a building block for a universal construction, we define the weak log object, a data structure used as a list of presence where a process that arrives registers. We first propose a universal construction based on consensus objects and a weak log object and then an implementation of the weak log using read/write registers and consensus objects. This proves that consensus is universal in all models considered in this paper.

Interestingly, this presentation also highlights how constructions that verify both liveness and safety conditions can be obtained as a combination of a seed of liveness, here illustrated by the wait-free weak log that is not linearizable but only eventually consistent, and a sprout of safety, in our case the list of operations that would be sufficient for a lock-free linearizable universal construction without the need of the weak log. It also illustrates a use case for weak consistency in a situation where strong consistency can also be achieved.

4.1 The weak log abstraction

We first define the weak log abstraction. In an instance of the weak log, a process p_i proposes a value through an operation `append`(v_i), that returns the sequence of all the values previously appended. The weak log is wait-free but not linearizable, in the sense that there might be no inclusion between the sequences returned by different operations. However, it is requested that values appended by correct processes will eventually appear in all returned sequences and that the order of the values within each sequence is consistent with the different sequences returned by all operations.

Definition 3 (*Weak log*) A process p_i proposes a value v_i (all appended values are assumed different) by invoking `append`(v_i), that returns a finite sequence $w_i = w_{i,1} \cdot w_{i,2} \cdots w_{i,|w_i|}$ such that:

Validity. Any value returned in a sequence was the argument of some invocation of `append`.

Suffixing. If some invocation of `append`(v_i) terminates, then v_i is appended at the end of its returned sequence w_i: $\forall i, w_{i,|w_i|} = v_i$.

Total order. If two invocations of `append` return respectively w_i and w_j, then all pairs of values that w_i and w_j both contain appear in the same order: for all i, j, k_i, k_j, l_i, l_j such that $w_{i,k_i} = w_{j,k_j}$ and $w_{i,l_i} = w_{j,l_j}$, we have $k_i \leq l_i$ if, and only if, $k_j \leq l_j$.

Eventual visibility. If some invocation of `append`(v_i) terminates, then, eventually, all sequences returned by invocations of `append` will contain v_i. In other words, the number of returned sequences that do not contain v_i is finite.

Wait-freedom. Any invocation of `append`(v_i) by a correct process p_i eventually returns.

4.2 A universal construction

Algorithm 1 presents a universal construction using a weak log and consensus objects. This algorithm is similar to the one

[5] A small guided tour on universal constructions can be found in [20].

[6] This means that any operation on the object can be called and the call returns regardless of the state of the object.

```
constructor (state_i) is
1   |  announce ← new WeakLog;
2   |  operations ← new Cons⟨ℕ⟩;
3   |  initialState ← state_i;
operation apply(invoc_i) is
4   |  toHelp_i ← announce.append(invoc_i);
5   |  cons_i ← operations;
6   |  state_i ← initialState;
7   |  while toHelp_i ≠ ε do
8   |  |  node_i ← new Node {value ← toHelp_i[0]; next ← new Cons⟨ℕ⟩};
9   |  |  node_i ← cons_i.propose(node_i);
10  |  |  cons_i ← node_i.next;
11  |  |  toHelp_i ← toHelp_i \ node_i.value;
12  |  |  r_i ← state_i.invoke(node_i.value);
13  |  |  if node_i.value = invoc_i then result_i ← r_i;
14  |  return result_i;
```

Algorithm 1: Wait-free universal construction in the infinite arrival model

presented in [14], except that the array of single-writer/multi-reader registers used by processes to announce their operations is replaced by a weak log. A universal construction emulates any shared object. The shared object to implement is represented by an initial state `initialState`, passed as an argument to the constructor, and a set of operations called invocations that change the state of the object and return a value. A process p_i that want to execute an operation $invoc_i$ on the emulated object calls `apply`($invoc_i$) on the universal construction.

Processes share two variables:

- `announce`is a weak log in which processes append their invocations;
- `operations`is a consensus object at the head of a linked list of operations. The list is a succession of nodes of type `Node`, defined as a structured type made up of two fields: `value` is the invocation of some process, and `next` is a consensus object referencing another node of type `Node` after the consensus has been won by some process.

When process p_i calls `apply`($invoc_i$), it first appends $invoc_i$ to `announce`and obtains in return a list $toHelp_i$ of invocations. Then, it attempts to insert the invocations of $toHelp_i$ at the end of the list `operations`until all the invocations of $toHelp_i$ have been inserted. While traversing the list, it maintains a state $state_i$ of the implemented object, initialized to `initialState` and on which all invocations are applied in their order of appearance in the list.

We now prove that Algorithm 1 is linearizable and wait-free. Linearizability is achieved by Algorithm 1 in the same way as in [14], so the proof of Lemma 1 is similar.

Lemma 1 (Linearizability) *All executions admissible by Algorithm* 1 *are linearizable.*

Proof Let α be an execution admissible by Algorithm 1.

Let us first remark that, for any operation `apply`($invoc_i$) invoked by process p_i, at most one node *node* is such that $node.\texttt{value} = invoc_i$. Indeed, suppose this is not the case, and let us consider the first two such nodes, $node_j$ and $node_k$. Both were proposed on line 9 by processes p_j and p_k respectively. As operations are totally ordered in a list, process p_k accessed $node_j$ before accessing $node_k$. After accessing $node_j$ and executing line 11, $invoc_i = node_j.\texttt{value} \notin toHelp_k$, which contradicts the fact that p_k proposed $node_k = invoc_i$.

Let us define the linearization point of any operation `apply`($invoc_i$) as, if it exists, the first step in which some process p_j proposed a node $node_j$ with $node_j.\texttt{value} = invoc_i$ and won the consensus on line 9.

We now prove that any operation `apply`($invoc_i$) done by a terminating process p_i has a linearization point, between its invocation and termination point. By the *validity* property of `announce`, and as all $invoc_i$ values are different, no process proposes $invoc_i$ before p_i arrived in the system.

By the *suffixing* property of `announce`, at the beginning of p_i's loop, $invoc_i \in toHelp_i$. When p_i terminates, $invoc_i \notin toHelp_i$. Therefore, $invoc_i$ was removed on line 11 of some iteration of the loop, so some process won a consensus where it proposed a node $node_j$ with $node_j.\texttt{value} = invoc_i$.

Finally, operations are applied by p_i on $state_i$ in the same order as they appear in the list (lines 12), which is the same order as their linearization points, which concludes the proof. □

The proof of wait-freedom (Lemma 2) is more challenging because the proof of [14] heavily relies on the fact that the number of processes is finite.

Lemma 2 (Wait-freedom) *All executions admissible by Algorithm* 1 *are wait-free.*

Proof Suppose there is an execution α admissible by Algorithm 1 that is not wait-free. It means that some process p_i takes an infinite number of steps in its invocation of `apply`($invoc_i$) in α. By the *wait-freedom* property of `announce`, p_i enters the while loop after a finite number of steps, and each iteration of the loop terminates. Therefore, p_i executes an infinite number of loop iterations. Let $w_i = w_{i,1} \cdot w_{i,2} \cdots w_{i,|w_i|}$ be the initial value of $toHelp_i$. Before Line 9, as w_i is finite and $node_i$.`value` equals some $w_{i,k}$ at each iteration, there exists a value k such that $node_i$.`value` $= w_{i,k}$ an infinite number of times.

Let $win_0, win_1, \ldots$ be the infinite sequence of the values taken by $node_i$.`value` just after Line 9 during the execution, let $p_{\omega(j)}$ be the process that took the step on line 9 installing the value win_j in the consensus and let $p_{a(j)}$ be the process that invoked `apply`(win_j).

As processes $p_{\omega(j)}$ always proposes the first invocation of $toHelp_{\omega(j)}$ that was not inserted in the list yet, there is an infinite number of values win_j such that either

- $w_{i,k}$ is not part of $w_{\omega(j)}$ or
- $w_{i,k}$ is part of $w_{\omega(j)}$, but appears after win_j in the list.

By the *eventual visibility* property, the first case only concerns a finite number of win_j, so there is an infinite number of values win_j in the second case.

For each of them, by the *suffixing* property, $win_j = w_{a(j),|w_{a(j)}|}$, i.e. the process that invoked `apply`(win_j) obtained win_j as the last value of its $toHelp_{a(j)}$. By the *total order* property, it is impossible that $p_{\omega(j)}$ obtains win_j before $w_{i,k}$ and $p_{a(j)}$ obtains $w_{i,k}$ before win_j. Therefore $w_{a(j)}$ does not contain $w_{i,k}$. However, this contradicts the *eventual visibility* property that prevents an infinite number of $p_{a(j)}$ processes to ignore $w_{i,k}$.

This contradicts the assumption of a non wait-free execution. □

4.3 An implementation of the weak log

The main difficulty in the implementation of a weak log lies in the allocation of one memory location per process, where it can safely append its value. As it is impossible to allocate an infinite array at once, it is necessary to build a data structure in which processes allocate their own piece of memory, and make it reachable to other processes, by winning a consensus. The list (`operations`) of Algorithm 1 displays such a pattern, but it poses a challenge: as an infinite number of processes access the same sequence of consensus objects, one process may loose all its attempts to insert its own node, breaking wait-freedom.

Algorithm 2 solves this issue by using a novel feature, that we call *passive helping*: when a process wins a consensus, it creates a side list to host values of processes concurrently competing on the same consensus object. As only a finite number of processes have arrived in the system when the consensus is won, a finite number of processes will try to insert their value in the side list, which ensures termination. Figure 2 presents an execution of Algorithm 2.

In other words, the processes share a main list of side lists of appended values. Side lists are a succession of nodes of type `SideNode`, defined as a structured type made up of two fields: `value` is a value appended by some process, and `next` is a consensus object referencing another node of type `SideNode` after the consensus has been won by some process. Similarly, the main list is a succession of nodes of type `MainNode`, defined as a structured type made up of two fields: `side` is a reference to a `SideNode`, and `next` is a consensus object referencing another node of type `MainNode` after the consensus has been won by some process.

Processes executing Algorithm 2 share two variables: `first` and `last` defined as follows.

- `first` is a consensus object on references to `MainNode`, at the beginning of the main list.
- `last` is a read/write register referencing a consensus object on references to `MainNode`. In the absence of concurrency, `last` references the `next` field of the last `MainNode` of the main list. Initially, the main list is empty and `last` is set to a reference at `first`.

When a process p_i invokes `append`(v_i), it first creates a `SideNode` own_i containing its value v_i. Then, it reads `last`, and proposes a new `MainNode` whose side list is only composed of own_i as its successor, and writes the `next` consensus field of the `MainNode` returned by the consensus in `last`. If p_i loses the consensus, it inserts own_i in the side list of the winner of the consensus (lines 7, 9 and 10). After that, p_i traverses the list of lists to build the sequence log_i it returns ($\oplus$ represents concatenation).

Note that the consensus and the write on lines 6 and 8 are not done atomically. This means that a very old value can be written in `last`, in which case its value could move backward. The central property of the algorithm, proved by Lemma 3, is that `last` eventually moves forward, allowing very slow processes to find some place in a side list.

Lemma 3 *If an infinite number of processes execute Line* 8*, then the number of processes that read the same* `lastvalue` *at Line* 6 *is finite.*

```
constructor () is
1   | first ← new Cons⟨ℕ⟩;
2   | last ← new R/W-Register(first);
operation append(v_i) is
    | // add v_i to the log
3   | own_i ← new SideNode {value ← v_i; next ← new Cons⟨ℕ⟩};
4   | main_i ← new MainNode {side ← own_i; next ← new Cons⟨ℕ⟩};
5   | cons_i ← last.read();
6   | main_i ← cons_i.propose(main_i);
7   | side_i ← main_i.side;
8   | last.write(main_i.next);
9   | while side_i ≠ own_i do
10  |   | side_i ← side_i.next.propose(own_i);
    | // read the log
11  | log_i ← ε; main_i ← first.get(); side_i ← main_i.side;
12  | while true do
13  |   | log_i ← log_i ⊕ side_i.value;
14  |   | if side_i = own_i then return log_i;
15  |   | side_i ← side_i.next.get();
16  |   | if side_i = ⊥ then
17  |   |   | main_i ← main_i.next.get();
18  |   |   | side_i ← main_i.side;
```

Algorithm 2: Wait-free weak log using consensus

Proof We first prove by induction on the succession of nodes in the main list of the weak log that for each `MainNode` *main*, the number of `write`operations of *main*.`next` in `last`at line 8 is finite.

– Initially, `first`is never written in `last`, because only decided values on line 6 are written, and `first`is never proposed.

– We now prove that, if the number of writes of *main* in `last`is finite, then the number of writes of *main*.`next` in `last`is finite.
We prove the following contrapositive proposition: if the number of writes of *main*.`next` in `last`is infinite, then the number of writes of *main* in `last`is infinite as well. In order to write *main*.`next` in `last`, a process needs to read *main* in `last`at line 6. As *main*.`next` is written an infinite number of times and *main* is read an infinite num-

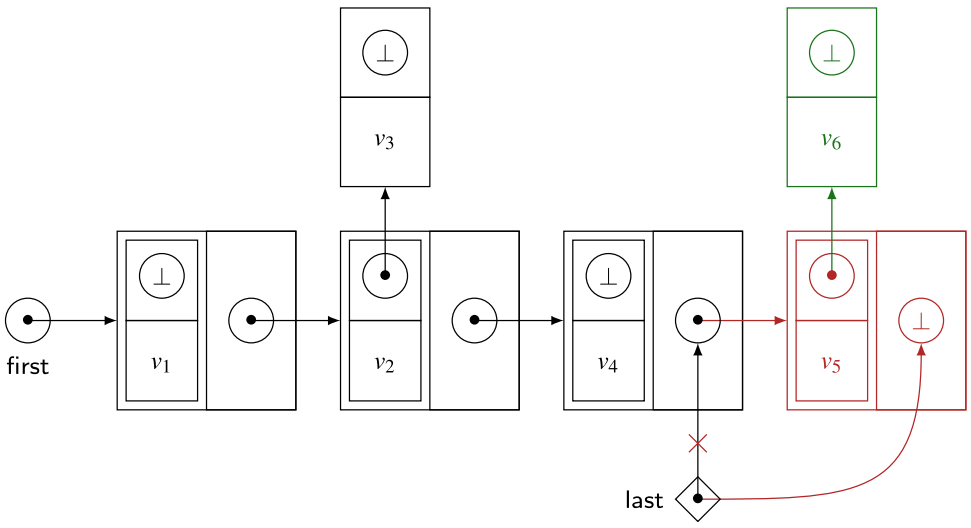


Fig. 2 An execution of Algorithm 2. Consensus objects and read/write registers are represented respectively with circles and diamonds. Processes p_5 and p_6 attempt to concurrently insert v_5 and v_6, respectively, in the weak log. They both read the same value in `last`, referencing the consensus object in the main list containing v_4. Process p_5 wins the consensus and inserts v_5 in its own side list after the one containing v_4, and p_6 loses the consensus, so it inserts v_6 in the side list created by p_5

ber of times, then necessarily, *main* is written an infinite number of times as well.

Let us now suppose that an infinite number of processes execute line 8, and that an infinite number of reads of `last`return *main*. This implies that there was an infinite number of `write`operations of *main* in `last`at line 8, which contradicts the previous induction result. □

Lemma 4 (Validity) *All the values in a returned sequence have been appended by processes.*

Proof The value log_i returned by the algorithm is built by concatenation of values $side_i$.`value` that can only be created, at line 6 or 10, using an appended value. □

Lemma 5 (Suffixing) *If w_i is the sequence returned when p_i appended v_i then $w_{i,|w_i|} = v_i$.*

Proof This is a direct consequence of the fact that $v_i = own_i$.`value` is appended at the end of log_i at Line 13 just before the return statement on Line 14. □

Definition 4 formalizes the order in which values are ordered in the weak log. Intuitively, this order is the concatenation of all the side lists, in the order of the main list. In Algorithm 2, the main list is traversed in this precedence order, which ensures consistency of the order of all returned sequences (*Total order* property of the weak log).

Definition 4 (*Precedence*) A `SideNode` s precedes a `SideNode` s' in the weak log if:

- there exists a sequence of `SideNode` $\{s_1, \ldots, s_n\}$ such that for all $1 \leq k < n$, s_{k+1} is decided in s_k.`next`, $s_1 = s$, and $s_n = s'$;
- or there exists a sequence of `MainNode` $\{m_1, \ldots, m_n\}$ such that for all $1 \leq k < n$, m_{k+1} is decided in m_k.`next`, m_1.`side` precedes s and m_n.`side` precedes s'.

A value v precedes a value v' in the weak log if there exist two `SideNode` s and s' such that s precedes s', s.`value` $= v$ and s'.`value` $= v'$.

Lemma 6 (Total order) *If two processes p_i and p_j terminate their invocations, then all pairs of values that belong to both w_i and w_j appear in the same order.*

Proof Let us remark that both processes p_i and p_j append values in their log following the precedence order defined by Definition 4. Therefore, for any two values v_k and v_l that appear in both w_i and w_j, p_i and p_j have appended them at the end of the log in the same order, which proves the lemma. □

Lemma 7 (Eventual visibility) *If some process p_i terminates its invocation, then the number of returned sequences that do not contain v_i is finite.*

Proof Let us denote by m_i and s_i the values of $main_i$ and $side_i$ when p_i terminates.

Let us suppose (by contradiction) that there is an infinite number of processes which return sequences that do not contain v_i, and an infinite number of them started their operation after p_i returned. For each such process p_j, let m_j and s_j be the values of $main_j$ and $side_j$ when p_j terminates its execution. As the collect loop respects the precedence order of Definition 4, for an infinite number of p_j, m_j precedes m_i. As there is only a finite number of lists preceding m_i (p_i terminates), an infinite number of processes have the same value m of m_j. All of them read the same value of `last`at Line 6 and wrote on Line 8. This contradicts Lemma 3. □

Lemma 8 (Wait-freedom) *No process takes an infinite number of steps in an execution.*

Proof Let us suppose that there exists an execution α such that process p_i takes an infinite number of steps in α trying to append v_i. This means that one of the two loops (lines 9 and 12) loops an infinite number of times:

- If the loop at Line 9 loops for an infinite number of times, it means that $side_i \neq own_i$ for an infinite number of nodes. This implies that an infinite number of values are appended to the same side list at Line 10, which means that an infinite number of processes read the same value at Line 6, and wrote at Line 8, which contradicts Lemma 3.
- If the loop at Line 12 loops forever, this means that p_i never reads v_i, and as there is a finite number of `MainNode` that precede the list m_i in which v_i has been appended, one of their side lists contains an infinite number of nodes. All these nodes were created by processes reading the same value of `last`at Line 6, which also contradicts Lemma 3.

Both cases lead to a contradiction. □

Theorem 1 *Multi-valued consensus has consensus number ∞_3^3.*

Proof By Lemmas 4, 5, 6, 7 and 8 , Algorithm 2 implements a weak log. By Lemmas 1 and 2 , Algorithm 1 is a universal construction in $M_3[\texttt{Cons}\langle\mathbb{N}\rangle]$. □

Remark 1 The usual algorithm for solving consensus using the compare-and-swap special instruction on atomic registers does not need any adaptation to work in model M_3. Therefore, compare-and-swap has consensus number ∞_3^3 as well.

5 Infinite memory allocation is not necessary in M_1

The original paper on the wait-free hierarchy [12] mentions no limitation that could arise in computing models where infinite allocation is not available. In this section, we prove that, in the context of bounded arrival models, infinite memory allocation is not a decisive factor to determine if universality can be achieved or not. This implies that our hierarchy is an extension of that proposed by Herlihy because they coincide for objects with a finite consensus number. This justifies our choice to keep the same name.

This result builds on the observation that, in MA_1^n, any wait-free algorithm of binary consensus has a bound on the number of memory locations used by any execution, as long as there is a bound on process identifiers (Lemma 9). Such a bound can be obtained by using renaming algorithms: for example, the algorithm introduced in [5] does not require infinite memory allocation either. In this section, we suppose, without loss of generality, that there is a bound N on process identifiers.

Lemma 9 *For any object O, if $\texttt{Cons}\langle\mathbb{B}\rangle$ can be implemented in $MA_1^n[O]$, then $\texttt{Cons}\langle\mathbb{B}\rangle$ can be implemented in $M_1^n[O]$.*

Proof Suppose there exists an algorithm A that implements binary consensus in $MA_1^n[O]$. As discussed in Sect. 3, we can suppose without loss of generality that A is a one-shot consensus, hence an input of A is composed of a set Π of at most n processes taken from $\{p_0, \ldots, p_N\}$, and a map that associates a Boolean input to each process in Π. The number of possible inputs is bounded by $2^N \times 2^n$.

For each possible input $\langle \Pi \subset \{p_0, \ldots, p_N\}, \Pi \to \mathbb{B}\rangle$, let us consider the tree of all possible executions of A with this input: the root is the empty execution, and an execution $\alpha\beta$ is the son of an execution α if β is a step taken by some process p_i according to A. By construction, the tree is locally finite since no execution can have more than n sons, and as A is wait-free, the graph does not contain any infinite path. Therefore, by Konig's lemma, for each possible input $\langle \Pi \subset \{p_0, \ldots, p_N\}, \Pi \to \mathbb{B}\rangle$, a finite number of configurations may be accessed by some execution.

Finally, a finite number X_n of configurations are accessible by any execution of A. In each configuration, each process may be about to invoke an operation on a different shared object, so at most a finite number $n \times X_n$ of objects can be used by A. Therefore, A can be simulated by an algorithm in $M_1^n[O]$ that only allocates $n \times X_n$ memory locations at set up. □

Theorem 2 *For any object O, if $MA_1^n[O]$ is universal, then $M_1^n[O]$ is also universal.*

Proof Suppose that $MA_1^n[O]$ is universal; by definition, $\texttt{Cons}\langle\mathbb{B}\rangle$ can be implemented in $MA_1^n[O]$. By Lemma 9, $\texttt{Cons}\langle\mathbb{B}\rangle$ can be implemented in $M_1^n[O]$. It is possible to implement $\texttt{Cons}\langle\mathbb{N}\rangle$ using a bounded number of $\texttt{Cons}\langle\mathbb{B}\rangle$ objects in the bounded arrival model using an algorithm like the one given in [23], and that can be easily adapted to shared memory [19]. Finally, by Theorem 1, O is universal in $M_1^n[O]$. □

Remark 2 Since $M_1[O]$ is the union of the $M_1^n[O]$ for all n, Theorem 2 implies that, if $MA_1[O]$ is universal, then $M_1[O]$ is also universal. However, this does not mean that $M_1[O]$ and $MA_1[O]$ are equivalent. In particular, some algorithms from MA_1 that use infinite arrays for other reasons than creating a universal construction or assigning one single-writer/multiple-reader register to each process [1,3] might not be possible to adapt to M_1.

6 No object has consensus number ∞_1^1

In this section, we prove that no object has consensus number ∞_1^1. We prove this by showing that, when infinite memory allocation is available, any universal object O in the bounded arrival model is also universal in the finite arrival model. Indeed, if $MA_1[O]$ is universal, it is possible to use objects O to solve consensus among n processes, for all n. Algorithm 3 then uses these $\texttt{Cons}^n\langle\mathbb{N}\rangle$ objects to solve Consensus in MA_2 (Lemmas 10, 11 and 12).

Processes share three infinite arrays: `greaterId`, `cons` and `adopt`. For each index $r \in \mathbb{N}$, `greaterId`$[r]$ is a Boolean register, initially **false**, that can be written by p_i only if $i \geq r$; `cons`$[r]$ is a $\texttt{Cons}^r\langle\mathbb{N}\rangle$ object that accepts participation of processes $p_0, \ldots, p_{r-1}$; and `adopt`$[r]$ is a register, initially $\perp$ (any value that cannot be proposed), that will store the decided value of `cons`$[r]$ so that processes $p_r, p_{r+1}, \ldots$ can know the decided value without participating.

Algorithm 3 is round-based. At round r, processes with identifiers smaller than r agree on some value using the $\texttt{Cons}^r\langle\mathbb{N}\rangle$ object `cons`$[r]$, while the other processes simply announce their presence by marking `greaterId`$[r]$. If the former decide first, they return the value they decided. Otherwise, if the latter arrive before consensus took place, more rounds are necessary. If the two groups write concurrently, it is possible that some processes decide a value at round r while others start round $r + 1$. In that case, the protocol ensures that they adopt the decided value for the next rounds, ensuring agreement.

```
constructor () is
1   | greaterId ← new InfiniteArray(r ↦ new R/W-Register(false));
2   | cons ← new InfiniteArray(r ↦ new Cons^r⟨ℕ⟩);
3   | adopt ← new InfiniteArray(r ↦ new R/W-Register(⊥));
operation propose(val_i) is
4   | v_i ← val_i;
5   | for r_i = 1,2,...,i do
6   |   | greaterId[r_i].write(true);
7   |   | x_i ← adopt[r_i].read();
8   |   | if x_i ≠ ⊥ then v_i ← x_i;
9   | for r_i = i+1, i+2, ... do
10  |   | v_i ← cons[r_i].propose(v_i);
11  |   | adopt[r_i].write(v_i);
12  |   | if ¬greaterId[r_i].read() then return v_i;
```

Algorithm 3: Consensus in Model $MA_2[\texttt{Cons}^n\langle\mathbb{N}\rangle\rangle]$ (code for p_i)

Claim For any round r, at most r processes invoke `propose` on `cons`$[r]$ Line 7.

Proof By Line 9, a process p_i can only execute Line 10 if $r > i$, and there are at most r processes with identifiers less than r. □

Lemma 10 (Wait-freedom) *All executions of Algorithm 3 terminate in MA_2.*

Proof In MA_2, each execution has a process with the greatest identifier (call it i_{max}). Variable `greaterId`$[r]$ is only set to **true** (Line 5) if $r \leq i_{max}$ (Line 4), so all processes terminate at the latest at round $i_{max}+1$ (Line 12). □

Lemma 11 (Validity) *If p_i decides v, then some process proposed v.*

Proof Let p_i be a process that decides v_i on round r_i. Let us suppose v_i is not the input of some process, and let us consider the first time a value that is not the input of some process is written in either v_j or `adopt`$[r_j]$ by some process p_j. This cannot happen on Line 4 by definition of val_j. By Claim 6 and validity for $\texttt{Cons}^{r_j}\langle\mathbb{N}\rangle$, only a value previously written in some v_j can be written in v_i on Line 10. Due to the condition $x_i \neq \perp$, only a value previously written in `adopt`$[r]$ can be written in v_i on Line 9, and only a value of v_i is written in `adopt`$[r_i]$ on Line 11. This is absurd, so the value v_i is set to val_i on Line 4 and is still a proposed value when p_i decides. □

Lemma 12 (Agreement) *If processes p_i and p_j decide respectively v_i and v_j, then $v_i = v_j$.*

Proof Let p_i be a process that terminates, deciding v_i, at the smallest round number r_i.

We first prove, by induction on r, that, for all $r > r_i$, all processes p_j participating to round r start the round with $v_j = v_i$. Suppose p_j participates at round $r = r_i + 1$. If $j > r_i$, then $v_j = v_i$ after p_j executed Line 10 during round r_i, by Claim 6 and the agreement property of $\texttt{Cons}^{r_i}\langle\mathbb{N}\rangle$. Otherwise, p_j set `greaterId`$[r_i]$ to **true** Line 6 after p_i read `greaterId`$[r_i]$ as **false** (Line 12), so p_j read `adopt`$[r_i]$ on Line 7 after p_i wrote v_i to `adopt`$[r_i]$ on Line 11. By Lemma 11, the value v_i decided by p_i was proposed by some process, so $v_i \neq \perp$, and p_j started round $r_i + 1$ with $v_j = v_i$ (Line 8). Let us suppose the claim holds for some $r > r_i$. By Claim 6 and the validity property of $\texttt{Cons}^r\langle\mathbb{N}\rangle$, only v_i can be decided on Line 10, and therefore written in `adopt`$[r]$ Line 11, so all processes either keep their value v_i (if the condition on Line 8 is false), or adopt the value v_i they read in `adopt`$[r]$ (Line 7) or the value v_i they decide on `cons`$[r]$ Line 10, to start round $r+1$.

Let p_j be a process that decides v_j at round $r_j \geq r_i$. We have $r_j > j$, so p_j returned the value decided on Line 10, which we have already established to be v_i. □

Theorem 3 *No object has consensus number ∞_1^1.*

Proof Suppose, by contradiction, that some object O has consensus number ∞_1^1. Hence, $MA_1[O]$ is universal. For all n, it is possible to implement a $\texttt{Cons}^n\langle\mathbb{N}\rangle$ object in MA_2, using O by simulating the algorithm of the bounded arrival model and setting the bound to n.

By Lemmas 10, 12 and 11 , Algorithm 3 is an implementation of consensus in MA_2, which is universal by Theorem 1. A contradiction. □

7 Objects with consensus number ∞_1^2

By Theorem 3, objects that have consensus number ∞_1^2 are the weakest objects that can be used to solve consensus among n processes, for all n, but are unable to adapt to an unknown number of processes. This section proves that set-or-decrement registers have consensus number ∞_1^2 (Theorem 4). Intuitively, this is because the number of processes

that can be synchronized using the `setOrDecrement` special instruction depends on the argument it is invoked with. The proof of Theorem 4 requires three intermediate results: the universality of set-or-decrement in M_1 (Proposition 1), and the impossibility to solve consensus in $MA_3[\mathtt{SOD}]$ (Proposition 2) and $M_2[\mathtt{SOD}]$ (Proposition 3).

7.1 Set-or-decrement is universal in M_1

Algorithm 4 presents an implementation of multi-valued consensus using a set-or-decrement register. The underlying idea is to encode each value v as the interval $[v \times n; v \times n + n - 1]$. The algorithm uses a set-or-decrement register `shared` initialized to 0. When a process p_i invokes `propose`(v_i), it invokes the operation `setOrDecrement` on `shared`, with the interval maximum $v \times n + n - 1$ as argument. If p_i is the first process to do so, it sets the value. Otherwise, it decrements the register by 1, which leaves the value of the register within the same interval. Finally, p_i reads the register to decode the decision value.

```
   constructor () is
1  |  shared ← new SOD(0);
   operation propose(v_i) is
2  |  shared.setOrDecrement(v_i × n + n − 1);
3  |  return ⌊ shared.read() / n ⌋;
```

Algorithm 4: Multi-valued consensus in $M_1[\mathtt{SOD}]$ (code for p_i)

Proposition 1 *$M_1[SOD]$ is universal.*

Proof Algorithm 4 is wait-free because it does not contain any loop. Let p_i be the first process that executes Line 2, writing $v_i \times n + n - 1$. After that, `shared` is decremented at most $n - 1$ times so all reads return a value between $v_i \times n + n - 1 - (n - 1) = v_i \times n$ and $v_i \times n + n - 1$ (Line 3) and all processes decide v_i. This implies validity and agreement on consensus, so `SOD` is universal. □

7.2 Set-or-decrement is not universal in MA_3

It was already noted in [2] that having access to $\mathtt{Cons}^n\langle\mathbb{N}\rangle$ objects for all n was not sufficient to solve consensus in MA_3. This section adapts the arguments to the `setOrDecrement` special instruction (Proposition 2). The proof relies on an extension of the classical notion of valency to runs that only contain steps by processes with identifiers smaller than n (Definition 5). In order to solve consensus between n processes, p_0 must reach an n-critical configuration (Lemma 13), in which it must invoke `setOrDecrement` with an argument larger than n (Lemma 14). In the infinite arrival model, more and more processes may arrive, forcing p_0 to invoke `setOrDecrement` with ever-growing arguments, breaking wait-freedom.

Definition 5 (*n-critical configuration*) Let α be an execution of a consensus algorithm. Let $C(\alpha)$ be the configuration (state of the computation) obtained after the execution α. We say that $C(\alpha)$ is v-n-valent if v can be decided in some extension $\alpha\beta$ of α in which only processes $p_0, p_1, \ldots p_{n-1}$ take steps. We say that $C(\alpha)$ is n-bivalent if it is both v-n-valent and w-n-valent for some $v \neq w$, and that it is v-n-univalent if it is v-n-valent and not n-bivalent. Finally, we say that $C(\alpha)$ is n-critical if it is n-bivalent and that the next step taken by any process in $p_0, p_1, \ldots p_{n-1}$ leads to a v-n-univalent configuration, for some v.

Lemma 13 *Any finite execution α such that $C(\alpha)$ is n-bivalent has an extension $\alpha\beta$ such that $C(\alpha\beta)$ is n-critical.*

Proof Suppose this is not the case. We build an infinite execution $\alpha' = \alpha\beta_1\beta_2\ldots$ such that, for all i, $\alpha_i = \alpha\beta_1\beta_2\ldots\beta_i$ leads to an n-bivalent configuration. For $i = 0$, $\alpha_0 = \alpha$ is an n-bivalent configuration. Suppose we have built such an execution for some i. By hypothesis, $C(\alpha_i)$ is not n-critical, so there is a process p_i whose next step is β_{i+1} such that $\alpha_{i+1} = \alpha_i\beta_{i+1}$ leads to an n-bivalent configuration. Finally, α' is infinite but finitely many processes arrived, so some process took an infinite number of steps, which contradicts wait-freedom. □

Lemma 14 *In any n-critical configuration, with $n > 2$, $p_0, \ldots, p_{n-1}$ are about to invoke `setOrDecrement`(x_i) with $x_i \geq n - 1$ on the same register, the value of which is non-positive.*

Proof Let $n > 2$ and let α be an execution leading to a n-critical configuration. Each process p_i is about to execute a step β_i on some shared object.

Let us suppose there exists a process p_i whose next step is a read. As $C(\alpha)$ is critical, $C(\alpha\beta_i)$ is v-n-univalent and there exists p_j such that $C(\alpha\beta_j)$ is w-n-univalent, with $v \neq w$. This is impossible since $C(\alpha\beta_i\beta_j)$ is v-n-univalent, but $C(\alpha\beta_j)$ and $C(\alpha\beta_i\beta_j)$ are indistinguishable to p_j.

Let us suppose that processes q_1 and q_2 are about to access two different registers x_1 and x_2. As $C(\alpha)$ is critical, these steps lead to v_1-n-univalent and v_2-n-univalent

configurations, and, if $v_1 = v_2$, there exists a process q_3 accessing a register x_3 and leading to a v_3-n-univalent configuration, with $v_3 \neq v_1 = v_2$. As $x_3 \neq x_1$ or $x_3 \neq x_2$, there always exist two processes p_i and p_j accessing different registers such that $C(\alpha\beta_i)$ is v-n-univalent and $C(\alpha\beta_j)$ is w-n-univalent, with $v \neq w$. This is impossible because $C(\alpha\beta_i\beta_j)$ is v-n-univalent, $C(\alpha\beta_j\beta_i)$ is w-n-univalent, and $C(\alpha\beta_i\beta_j) = C(\alpha\beta_j\beta_i)$.

Let us suppose that all processes are about to access the same register and there exists a process p_i whose next step is a write. As $C(\alpha)$ is critical, $C(\alpha\beta_i)$ is v-n-univalent and there exists p_j such that $C(\alpha\beta_j)$ is w-n-univalent, with $v \neq w$. This is impossible since $C(\alpha\beta_j\beta_i)$ is w-n-univalent, but $C(\alpha\beta_i)$ and $C(\alpha\beta_j\beta_i)$ are indistinguishable to p_i.

Let us suppose that all processes $p_0, \ldots, p_{n-1}$ are about to invoke $\texttt{SOD}(x_i)$ on the same register, whose value is $x > 0$. As $C(\alpha)$ is critical, there exist two processes p_i and p_j such that $C(\alpha\beta_i)$ is v-n-univalent and $C(\alpha\beta_j)$ is w-n-univalent, with $v \neq w$. As $n > 2$, there exists a third process p_k. This is impossible because $C(\alpha\beta_i)$ and $C(\alpha\beta_j)$ are indistinguishable to p_k, since the value of the register is $x - 1$ in both configurations.

Let us suppose that all processes $p_0, \ldots, p_{n-1}$ are about to invoke $\texttt{SOD}(x_i)$ on the same register whose value is nonpositive, and for some i, $x_i \leq n-2$. As $C(\alpha)$ is critical, $C(\alpha\beta_i)$ is v-n-univalent and there exists p_j such that $C(\alpha\beta_j)$ is w-n-univalent, with $v \neq w$. Let γ be some concatenation of the next step of x_i processes, excluding p_i and p_j. $C(\alpha\beta_i\gamma\beta_j)$ is v-n-univalent and $C(\alpha\beta_j)$ is w-n-univalent, but the two configurations are indistinguishable to p_j.

The only remaining case is that processes $p_0, \ldots, p_{n-1}$ are about to invoke $\texttt{SOD}(x_i)$ with $x_i \geq n-1$ on the same register, whose value is nonpositive. □

Proposition 2 *$MA_3[SOD]$ is not universal.*

Proof Suppose there exists an algorithm A that solves consensus in $MA_3[\texttt{SOD}]$. We build a sequence of executions $\alpha_0 = \beta_0, \alpha_1 = \beta_0\beta_1, \alpha_2 = \beta_0\beta_1\beta_2, \ldots$ and a sequence of integers $n_0 \leq n_1 \leq n_2 \leq \cdots$ such that, for all i, process p_0 takes a step in β_i and $C(\alpha_i)$ is n_i-critical.

For $i = 0$, let $n_0 = 3$, and γ be the execution in which each p_i proposes i. In a p_i-solo extension of γ, p_i decides i, so $C(\gamma)$ is n_0-bivalent. By Lemma 13, there is an extension α_0 of γ such that $C(\alpha_0)$ is n_0-critical.

Suppose we have built an execution α_i and an integer n_i respecting the induction invariant for some i. By Lemma 14, in $C(\alpha_i)$, $p_0, \ldots, p_{n_i-1}$ are about to invoke $\texttt{SOD}(x_i)$ on the same non-positive register, with $x_i \geq n_i - 1$. Let us pose $n_{i+1} = \max_{i \in \{0,\ldots,n_i-1\}} x_i + 2$.

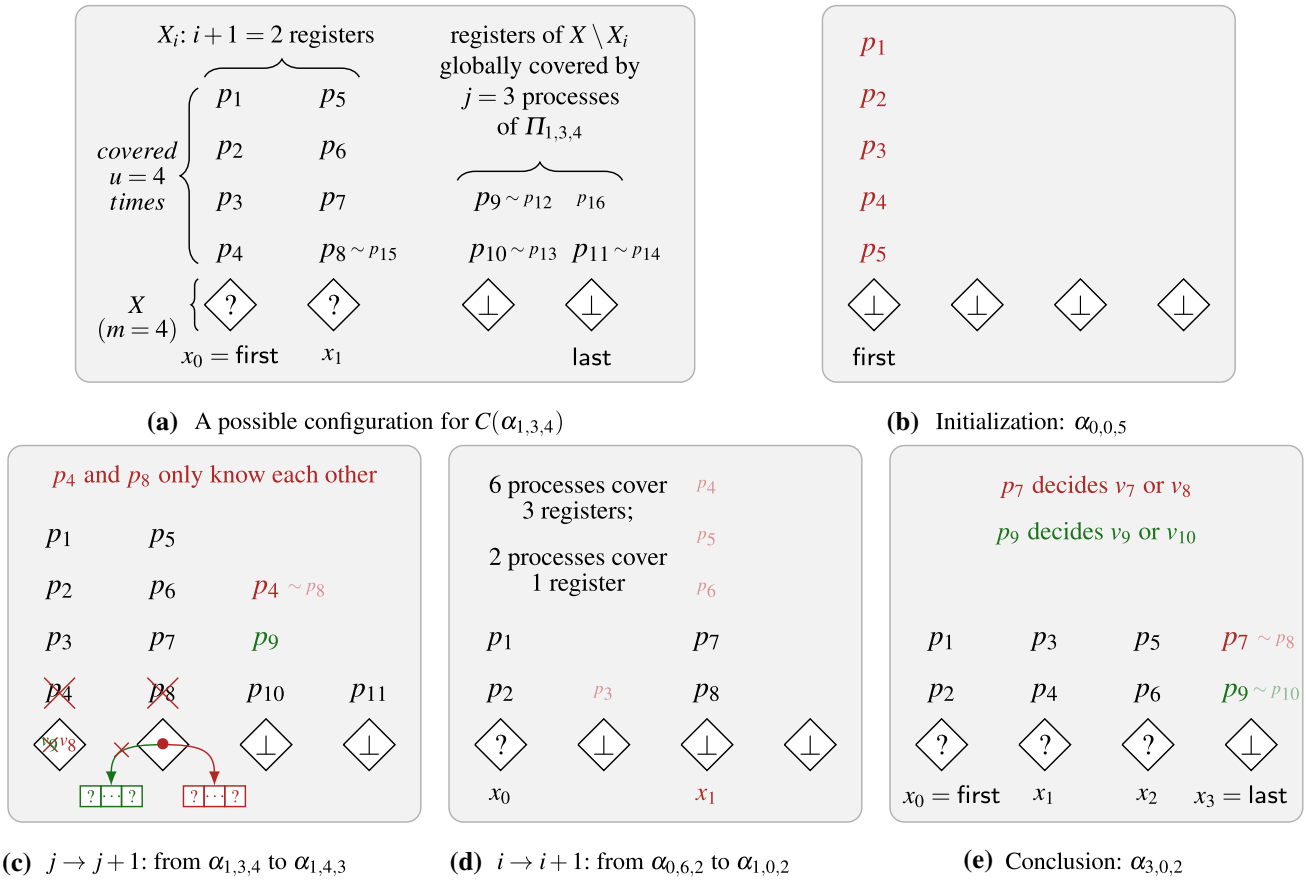
As $C(\alpha_i)$ is n_i-critical, $C(\alpha_i)$ is also n_i-bivalent, so $C(\alpha_i)$ is n_{i+1}-bivalent. By Lemma 13, there is an extension $\alpha_{i+1} = \alpha_i\beta_{i+1}$ of α_i such that $C(\alpha_{i+1})$ is n_{i+1}-critical. By Lemma 14, in $C(\alpha_{i+1})$, p_0 is about to invoke $\texttt{SOD}(x'_0)$, with $x'_0 \geq n_{i+1}-1 = \max_{i \in \{0,\ldots,n_i-1\}} x_i + 2 - 1 \geq x_0 + 2 - 1 > x_0$. In particular, p_0 took a step to invoke $\texttt{SOD}(x_0)$ in β_{i+1}.

To conclude, p_0 took an infinite number of steps in $\alpha = \beta_1\beta_2\ldots$, i.e. α is not wait-free. A contradiction. □

7.3 Set-or-decrement is not universal in M_2

Proposition 3 below shows that set-or-decrement registers are not universal in the finite arrival model when infinite memory allocation is not possible. In addition to set-or-decrement registers, one reason why infinite memory allocation mechanisms may be necessary and sufficient in M_2 is that the number of instances of set-or-decrement registers required by the synchronization grows boundlessly with the number of processes. Recently, [7] introduced a complexity-based hierarchy ranking shared objects according to the number of instances that are needed to solve obstruction-free consensus. For example, at least $O\left(\sqrt{n}\right)$ registers in addition to a test-and-set operation are necessary to solve obstruction-free multi-valued consensus between n processes. In order to be universal in M_2, an object has only two ways to circumvent the limitation that only a fixed and finite number of objects can be created at the initialization of any algorithm: either it has a constant complexity in the hierarchy proposed in [7], or it provides enough synchronization power to maintain an extensible data structure (e.g. a linked list), where new instances of itself can be created at runtime and accessed by newly arrived processes.

The proof of Proposition 3 has the same flavor as the proofs in [7], but simplified as we are only interested in decidability whereas their bounds need to be as tight as possible. Figure 3 illustrates the main steps of the proof. More precisely, the proof of Proposition 5 builds a scheduler that keeps track of a subset Π' of processes that have never communicated with each other because the values they write in registers are overwritten. The property maintained by the executions produced by this scheduler, called Π'-partitioning, is specified in Definition 6. The scheduler builds an execution in which a large number of processes participate, and more and more shared registers are covered by many processes (i.e. these processes try to write or invoke set-or-decrement in the registers, see Definition 7) ignore the existence of each other, until all objects are covered and two processes decide different values. The main difficulty of the proof is that the number of processes that need to participate depends on the arguments of the `setOrDecrement` invocations, so it has to be guessed by the scheduler before the information is available. Nevertheless, due to Π'-partitioning, processes in different partitions cannot communicate to adapt their arguments to the number of processes, which allows us to pick the appropriate number of processes *a posteriori*.



(a) A possible configuration for $C(\alpha_{1,3,4})$

(b) Initialization: $\alpha_{0,0,5}$

(c) $j \to j+1$: from $\alpha_{1,3,4}$ to $\alpha_{1,4,3}$

(d) $i \to i+1$: from $\alpha_{0,6,2}$ to $\alpha_{1,0,2}$

(e) Conclusion: $\alpha_{3,0,2}$

Fig. 3 Illustration of the covering arguments used in the paper, by a proof that M_2 is not universal. The goal is to build an execution such that two processes are about to terminate but do not know about each other, hence deciding a different value (Fig. 3e). We do that by building, inductively, an execution in which more and more registers (pictured in diamond shapes) are covered enough times (Fig. 3d, starting with the register `first` (Fig. 3b). The difficult part in most proofs is to make one more process cover one more register (Fig. 3c). Here, two processes (p_4 and p_8) have overwritten the values written by p_9 in x_0 and x_1 and been executed in isolation until p_4 covers a register in $X \setminus X_1$. Notice that p_4 and p_8 may have learned about each other in the process, but not about p_9. Hence, we maintain a set $\Pi_{i,j,u}$ of processes that do not know about each other. Other processes (in light color), are ignored in the rest of the execution, but they might only be known by one process still in $\Pi_{i,j,u}$, which we encode by an equivalence relation $\sim_{i,j,u}$. The notations used in the proofs are illustrated on Fig. 3a

Definition 6 (*Partitioned execution*) Let $\Pi' \subset \Pi$ be a set of processes, let $\sim$ be an equivalence relation on Π, and let $p \in \Pi'$ be a process. We say that a finite execution α is $(\Pi', \sim, p)$-partitioned (or simply Π'-partitioned if $\sim$ and p are immaterial) if: (1) for all processes $q, q' \in \Pi'$ $q \nsim q'$, (2) for all processes $q \in \Pi'$, the restriction α_q of α to steps taken by processes $q' \sim q$ is a valid execution of the algorithm, and (3) all shared registers have the same value in $C(\alpha)$ and $C(\alpha_p)$.

Definition 7 (*Covered register*) Let p be a process and x be a register, we say that p covers x in a configuration C if the next step performed by p in C is either a write to x or an invocation of `setOrDecrement` on x.

Proposition 3 *$M_2[SOD]$ is not universal.*

Proof Let us suppose there exists an algorithm A that solves consensus in $M_2[\texttt{SOD}]$. To simplify the proof, we also suppose that processes start the algorithm by writing their value to some register `first`, and finish it by writing their decided value in another register `last`. Remark that such registers and steps can be added to any consensus algorithm without loss of generality. A finite set X of $m = |X|$ registers are created by the constructor of A.

We build (by induction on i and j), for all $i \in \{0, \ldots, m-1\}$, a shared register $x_i \in X$ (we define $X_i = \{x_0, \ldots, x_i\}$) and for all $j \in \mathbb{N}$ and $u \geq 2$, a set $\Pi_{i,j,u}$ of processes, and an execution $\alpha_{i,j,u}$ such that:

- $\alpha_{i,j,u}$ is $\Pi_{i,j,u}$-partitioned (we denote by $\sim_{i,j,u}$ the corresponding equivalence relation),
- all registers of at X_i are covered at least u times by processes in $\Pi_{i,j,u}$,

- j processes of $\Pi_{i,j,u}$ cover registers in $X \setminus X_i$,
- all registers of $X \setminus X_i$ are in their initial state,
- for all $u' > u$, $\Pi_{i,j,u} \subset \Pi_{i,j,u'}$ and for all processes $p \in \Pi_{i,j,u}$, $C(\alpha_{i,j,u})$ and $C(\alpha_{i,j,u'})$ are indistinguishable to p. In particular, it means that the classes of equivalence of $\sim_{i,j,u}$ are also classes of equivalence of $\sim_{i,j,u'}$.

Initialization for $i = 0$ and $j = 0$. Let $u \geq 2$. We pose $x_0 = \texttt{first}$, $\Pi_{0,0,u} = \{p_1, \ldots, p_u\}$ and $\sim_{0,0,u}$ as the equality over $\Pi_{0,0,u}$. In execution $\alpha_{0,0,u}$, each process $p_k \in \Pi_{0,0,u}$ proposes its own identifier k and stops executing when it is about to write in $x_0 = \texttt{first}$. Execution $\alpha_{0,0,u}$ is $\Pi_{0,0,u}$-partitioned because no process accessed any shared object, and all u processes of $\Pi_{0,0,u}$ cover x_0 by construction.

From j to $j+1$, for a fixed i. Suppose that, for some $i \in \{0, \ldots, m-1\}$, we have built X_i, and for some $j \in \mathbb{N}$, we have built, for all $u \geq 2$, $\Pi_{i,j,u}$ and $\alpha_{i,j,u}$ verifying the properties stated above.

Let us first remark that $\texttt{last} \notin X_i$. Otherwise, at least 2 processes q and q' of $\Pi_{i,j,2}$, would cover $\texttt{last}$ in $C\left(\alpha_{i,j,2}\right)$. Let α' be the extension of $C\left(\alpha_{i,j,2}\right)$ in which q, then q', writes in $\texttt{last}$, and the decide the same value v by agreement of A. By validity of A, and as all processes proposed different values, some unique process q_v proposed v. Since α' is partitioned, q also decided v in α'_q, so by validity of A again, $q_v \sim_{i,j,2} q$. Similarly, $q_v \sim_{i,j,2} q'$, so $q \sim_{i,j,2} q'$, which contradicts the fact that q and q' both belong to $\Pi_{i,j,2}$.

For all registers $x \in X_i$, let $\bar{x}$ be the value stored in x in Configuration $C(\alpha_{i,j,2})$. We pose $U = \max_{x \in X_i}(\max(\bar{x}, 0)) + 2$. Let us also pick a subset $\Phi_{i,j}$ of $\Pi_{i,j,U}$, containing, for each $x \in X_i$, either:

- one process about to write in x, or
- one process about to invoke $\texttt{setOrDecrement}$ on x if $\bar{x} \leq 0$, or
- $\bar{x} + 1$ processes about to invoke $\texttt{setOrDecrement}$ on x,

in Configuration $C(\alpha_{i,j,U})$. Such a $\Phi_{i,j}$ exists because $U \geq 2$ processes of $\Pi_{i,j,U}$ covered each register of X_i. Moreover, for all $u \geq 2$, $\Phi_{i,j} \subset \Pi_{i,j,u+U}$. Let us pick $p_{i,j} \in \Phi_{i,j}$.

Let $u \geq 2$. We pose $\Pi_{i,j+1,u} = (\Pi_{i,j,u+U} \setminus \Phi_{i,j}) \cup \{p_{i,j}\}$, $\sim_{i,j+1,u}$ as the equivalence relation built by merging the classes of equivalence of processes in $\Phi_{i,j}$ in $\sim_{i,j,u+U}$, and $\alpha_{i,j+1,u} = \alpha_{i,j,u+U}\beta_{i,j,u}$, where $\beta_{i,j,u}$ is built by first letting each process $q \in \Phi_{i,j}$ take one step, and then executing $p_{i,j}$ until it covers a register in $X \setminus X_i$. Such a situation must happen because A is wait-free, and $p_{i,j}$ cannot terminate its execution before covering $\texttt{last} \in X \setminus X_i$.

All registers $x_u \in X_i$ were overwritten by processes in $\Phi_{i,j}$ before $p_{i,j}$ had a chance to do a read, and all registers in $X \setminus X_i$ are in their initial state, so $p_{i,j}$ only read values written by processes $p' \sim_{i,j+1,u} p_{i,j}$. Therefore, $\alpha_{i,j+1,u}$ is $\Pi_{i,j+1,u}$-partitioned. Moreover, all registers in X_i are still covered by at least u processes in $\Pi_{i,j,u+U} \setminus \Phi_{i,j}$. Adding $p_{i,j}$ to the j processes that already covered registers from $X \setminus X_i$ in $C(\alpha_{i,j,u})$, at least $j+1$ processes cover registers that are not in X_i, in $C(\alpha_{i,j+1,u})$. Moreover for $u' > u$, the executions $\alpha_{i,j+1,u'}$ and $\alpha_{i,j+1,u}$ are indistinguishable to all processes in $\Pi_{i,j+1,u}$, because $\alpha_{i,j,u'+U}$ and $\alpha_{i,j,u+U}$ are indistinguishable to them, and the same processes in $\Phi_{i,j}$ took the same steps in $\beta_{i,j,u}$ and $\beta_{i,j,u'}$.

From i to $i+1$ Suppose that, for some $i \in \{0, \ldots, m-2\}$, we have built X_i, and for all $j \in \mathbb{N}$ and $u \geq 2$, $\Pi_{i,j,u}$ and $\alpha_{i,j,u}$ verifying the properties stated above. Let $u \geq 2$. We pose $\Pi_{i+1,0,u} = \Pi_{i,(m-i)u,u}$ and $\alpha_{i+1,0,u} = \alpha_{i,(m-i)u,u}$. As $(m-i)u$ processes cover some of the $m-i$ registers in $X \setminus X_i$, by the pigeon holes theorem, at least one register, that defines x_{i+1}, is covered by at least u processes of $\Pi_{i+1,0,u}$. All registers of X_i are covered by the same u processes in $C(\alpha_{i+1,0,u})$ and in $C(\alpha_{i,(m-i)u,u})$. The other properties of $\alpha_{i+1,0,u}$ are naturally deduced from the properties of $\alpha_{i,(m-i)u,u}$.

Contradiction Finally, in $C(\alpha_{m-1,1,2})$, some process covers a shared object in $X \setminus X_i = \emptyset$ by definition of m. This is absurd, so A cannot exist. □

Theorem 4 *$\texttt{setOrDecrement}$ has consensus number ∞_1^2.*

Proof Direct consequence of Propositions 1, 2 and 3 . □

8 Objects with consensus number ∞_1^3

As advocated in Sect. 6, binary consensus has long been known to be equivalent to multi-valued consensus in the classical model M_1. This section extends this result by presenting an implementation of multi-valued consensus from binary consensus in the infinite arrival model, when an infinite memory allocation mechanism is available (Proposition 4). Conversely, we also prove that binary consensus is not universal in multi-threaded systems, because infinite memory allocation is necessary to solve multi-valued consensus in the finite arrival model (Proposition 5). Interestingly, this means that $\texttt{Cons}\langle\mathbb{B}\rangle$ is not at the top of the extended wait-free hierarchy, as it only has consensus number ∞_1^3 (Theorem 5).

8.1 Binary consensus is universal in MA_3

The sticky bit object, a resettable version of binary consensus, has been shown to be universal in MA_1 in [18]. Reductions of multi-valued consensus to binary consensus have later been proposed for message-passing systems [17], and extended to M_1 [19]. Algorithm 5 extends this result to the model MA_3. Processes share three infinite arrays: $\texttt{propose}$, $\texttt{isSet}$ and $\texttt{cons}$. For each index $j \in \mathbb{N}$, $\texttt{propose}[j]$ is intended to store the value proposed by p_j, $\texttt{isSet}[j]$ is a Boolean set

to true only after propose[j] has been set, and cons[j] is a binary consensus object in which **true** is decided if, and only if, the value of p_j is decided. When a process p_i proposes a value val_i, it first writes it to proposed[i] and sets isSet[i] to **true** to announce its value. Then, it browses the array indexes in the increasing order of the identifiers, trying to agree with other participants whether or not proposed[j] can be decided.

```
constructor () is
1   proposed ← new InfiniteArray(i ↦ new R/W-Register(⊥));
2   isSet ← new InfiniteArray(i ↦ new R/W-Register(false));
3   cons ← new InfiniteArray(i ↦ new Cons⟨B⟩);
operation propose(val_i) is
4   proposed[i].write(val_i);
5   isSet[i].write(true);
6   for j = 0,1,2,... do
7       set_i ← isSet[j].read();
8       if cons[j].propose(set_i) then
9           return proposed[j].read();
```

Algorithm 5: Consensus in $MA_3[\texttt{cons}\langle\mathbb{B}\rangle]$ (code for p_i)

Proposition 4 *$MA_3[Cons\langle\mathbb{B}\rangle]$ is universal.*

Proof We prove that Algorithm 5 implements Cons⟨$\mathbb{N}$⟩ in $MA_3[\texttt{Cons}\langle\mathbb{B}\rangle]$.

Termination. As all processes write **true** to isSet[i] (Line 5) before reading isSet[j] (Line 7), the first access to isSet is a write by some process p_{j_0}. All processes p_i executing the loop for $j = j_0$ will propose **true**, so by the validity property of binary consensus, no process executes an iteration for $j > j_0$.

Agreement. By the agreement property of binary consensus, all deciding processes decide on the same round j, which is the smallest x such that **true** is decided by cons[x].

Validity. Suppose p_i decides at round j on Line 9. Some process read isSet[j] = **true** on Line 7, so process p_j previously wrote **true** on Line 5, after writing its proposed value in proposed[r_j] on Line 4. This is the value returned by p_i.

Finally, by Theorem 1, $MA_3[\texttt{Cons}\langle\mathbb{N}\rangle]$ is universal. □

8.2 Binary consensus is not universal in M_2

Although Algorithm 5 solves consensus in the infinite arrival model, it requires $O(n)$ memory locations to synchronize n processes. Similarly, to our knowledge, no known algorithm uses less than $\log_2(n)$ binary consensus objects to solve multi-valued consensus in M_1^n in the worst case [23]. In this section, we prove that infinite memory allocation is necessary to make binary consensus universal in the finite arrival model. Proposition 5 below actually shows a more general result, stating that no deterministic object that can be in a finite number of states (which is the case for binary consensus), is universal in the finite arrival model without infinite memory allocation. Similarly to the proof of Proposition 3, the proof of Proposition 5 proposes a scheduler that builds executions in which all shared registers are covered by enough processes to force two of them to decide different values (the definition of covered register is adapted in Definition 8). Contrastingly, the proof of Proposition 5 differs from the proof of Proposition 3 in that the indistinguishability arguments concern the values proposed by the different participants (captured by a notion of *valuation*), rather than which processes participate.

Definition 8 (*Covered register*) Let p be a process and x an shared register. We say that p covers x in a configuration C if the next step of p in C is a write to x.

Proposition 5 *For all deterministic objects O with a finite number of reachable states, $M_2[O]$ is not universal.*

Proof Let O be a finite deterministic object, and let us suppose there exists an algorithm A that solves consensus in $M_2[O]$. Similarly to Proposition 3, we suppose, to simplify the proof, that processes start the algorithm by writing their value to some register first, and finish it by writing their decided value in another register last, not used otherwise in the algorithm. At the initialization of A, a finite set X of $m = |X|$ registers is created (we do not include instances of O in X, but their number must be finite in M_2).

For all $i \in \{0, \ldots, m\}$, we pose $u_i = 2(m - i + 1)!$. We consider executions of processes in the finite set $\Pi = \{p_1, \ldots, p_{u_0}\}$.

We will now build, by induction on $(i, j) \in \mathbb{N}^2$, with $0 \leq i < m$ and $0 \leq j \leq (m-i)u_{i+1}$, taken in lexicographical order:

- A sequence $x_i \in X$ of shared registers (we define $X_i = \{x_0, \ldots, x_i\}$);
- A sequence $\Pi_{i,j}$ of sets of processes;
- A sequence $\mathcal{V}_{i,j}$ of infinite sets of integers (we define a *valuation* V of $\mathcal{V}_{i,j}$ as a function that associates some $V(p) \in \mathcal{V}_{i,j}$ to each $p \in \Pi_{i,j}$, and for all $v \in \mathcal{V}_{i,j}$, let V_v be the constant valuation that associates v to each $p \in \Pi_{i,j}$);
- For each valuation V of $\mathcal{V}_{i,j}$, one execution $\alpha_{i,j}^V$;

Such that, for all valuations V, V' of $\mathcal{V}_{i,j}$:

IH1$_{i,j}$: All processes of $\Pi_{i,j}$ cover the same registers in $C\left(\alpha_{i,j}^V\right)$ and $C\left(\alpha_{i,j}^{V'}\right)$;

IH2$_{i,j}$: All instances of O are in the same state in $C\left(\alpha_{i,j}^V\right)$ and $C\left(\alpha_{i,j}^{V'}\right)$;

IH3$_{i,j}$: All registers of X_i are covered at least $(u_i - j)$ times in $C\left(\alpha_{i,j}^V\right)$;

IH4$_{i,j}$: At least j processes of $\Pi_{i,j}$ cover shared registers of $X \setminus X_i$ in $C\left(\alpha_{i,j}^V\right)$;

IH5$_{i,j}$: All registers of $X \setminus X_i$ are in their initial state in $C\left(\alpha_{i,j}^V\right)$;

IH6$_{i,j}$: For each process $p \in \Pi_{i,j}$, $C\left(\alpha_{i,j}^V\right)$ and $C\left(\alpha_{i,j}^{V_{V(p)}}\right)$ are indistinguishable to p.

Initialization for $i = 0$and $j = 0$ Let us pose $x_0 =$ `first`, $\Pi_{0,0} = \Pi$, and $\mathcal{V}_{0,0} = \mathbb{N}$. For each valuation V of $\mathcal{V}_{0,0}$, let $\alpha_{0,0}^V$ be the execution in which each process $p \in \Pi$ proposes $V(p)$ and stops executing when it is about to write into $x_0 =$ `first`. By construction, all processes of $\Pi_{0,0}$ cover `first` (hence, IH1$_{0,0}$), so x_0 is covered u_0 times (hence, IH3$_{0,0}$), and O is in the initial state regardless of the valuation (hence, IH2$_{0,0}$). No process covered or wrote any other register (hence, IH4$_{0,0}$ and IH5$_{0,0}$). Moreover, for all p and V, $C\left(\alpha_{i,j}^V\right)$ and $C\left(\alpha_{i,j}^{V_{V(p)}}\right)$ are indistinguishable to p since the processes did not communicate (hence, IH6$_{0,0}$).

From $j$to $j+1$,for a fixed i Suppose that, for some $i < m$ and for some $j < (m-i)u_{i+1}$, we have built X_i, $\Pi_{i,j}$, $\mathcal{V}_{i,j}$ and $\alpha_{i,j}^V$ for all V, verifying the induction hypotheses IH1$_{i,j}$ to IH6$_{i,j}$.

Let us remark that `last` $\notin X_i$. Otherwise, by IH3$_{i,j}$, at least $u_i - j > 2$ processes would cover `last` in $C\left(\alpha_{i,j}^V\right)$ for all valuations V of $\mathcal{V}_{i,j}$. Let us take p and q amongst them, and let V be an injective valuation of $\mathcal{V}_{i,j}$ (V exists because $\Pi_{i,j}$ is finite and $\mathcal{V}_{i,j}$ is infinite). Let α_V, α_p and α_q be respectively the extensions of $\alpha_{i,j}^V$, $\alpha_{i,j}^{V_{V(p)}}$ and $\alpha_{i,j}^{V_{V(q)}}$, in which p and then q, took their last step and decided a value. In α_p, all processes would have proposed $V(p)$ so p would decide $V(p)$ by the validity property of consensus. By IH6$_{i,j}$, α_p and α_V are indistinguishable to p, so p would decide α_V. Similarly, q would decide $V(q)$ in α_q and in α_V, which would violate the Agreement property of consensus.

Let us pick, arbitrarily, a set $\Phi \subset \Pi_{i,j}$ of $i+1$ processes, each covering a different register from X_i in $C\left(\alpha_{i,j}^V\right)$ (regardless of V, by IH1$_{i,j}$). Φ exists because of IH3$_{i,j}$, with $u_i - j > u_i - (m-i)u_{i+1} > 1$. Let us also pick, arbitrarily as well, a process $p \in \Phi$. We pose $\Pi_{i,j+1} = (\Pi_{i,j} \setminus \Phi) \cup \{p\}$.

For all valuations $\tilde{V}$ of $\mathcal{V}_{i,j}$ such that, for all $q \in \Phi$, $\tilde{V}(q) = \tilde{V}(p)$, we build $\tilde{\alpha}_{i,j}^{\tilde{V}}$ as the extension of $\alpha_{i,j}^{\tilde{V}}$ in which, at first, each process $q \in \Phi$ takes one step, which overwrites all registers in X_i ; then, p takes steps until it covers a register that is not in X_i. Such a situation must happen because, on the one side, A is wait-free so p cannot run in isolation forever, and on the other side, p must write into `last` $\notin X_i$ before terminating.

On the one hand, since $\mathcal{V}_{i,j}$ is infinite, there is an infinite set S of valuations on $\mathcal{V}_{i,j}$ that are constant on Φ. On the other hand, there are a finite number of ways for p to cover a register not in X_i and finitely many instances of O with finitely many states. Therefore, by the pigeon holes Theorem, there exists an infinite subset S' of S such that, for all valuations $\tilde{V}$ and $\tilde{V}'$ of S' that are constant on Φ, p covers the same register, and all instances of O are in the same state, in $C\left(\tilde{\alpha}_{i,j}^{\tilde{V}}\right)$ and $C\left(\tilde{\alpha}_{i,j}^{\tilde{V}'}\right)$. Let us pose $\mathcal{V}_{i,j}$ as the union of the ranges of the valuations in S'.

For all valuations V of $\mathcal{V}_{i,j+1}$, we define $\tilde{V}$ as the valuation of $\mathcal{V}_{i,j}$ such that $\tilde{V}(q) = V(q)$ for all $q \in \Pi_{i,j}$ and $\tilde{V}(q) = V(p)$ for all $q \in \Phi$, and we let $\alpha_{i,j+1}^V = \tilde{\alpha}_{i,j}^{\tilde{V}}$.

Let V and V' be two valuations of $\mathcal{V}_{i,j+1}$. By construction, and by IH1$_{i,j}$ and IH2$_{i,j}$, all processes of $\Pi_{i,j}$ cover the same registers, and all instances of O are in the same states in $C\left(\alpha_{i,j+1}^V\right)$ and $C\left(\alpha_{i,j+1}^{V'}\right)$, hence IH1$_{i,j+1}$ and IH2$_{i,j+1}$. Moreover, all registers in X_i are still covered by the processes in $\Pi_{i,j} \setminus \Phi$,

hence IH3$_{i,j+1}$. Adding p to the j processes that already covered registers from $X \setminus X_i$ in $C\left(\alpha_{i,j}^{\tilde{V}}\right)$ (by IH4$_{i,j}$), $j+1$ processes cover registers that are not in X_i, in $C\left(\alpha_{i,j+1}^V\right)$, hence IH4$_{i,j+1}$. Registers not in X_i were in their initial state in $\alpha_{i,j}^{\tilde{V}}$ by IH5$_{i,j}$, and were not overwritten in $\alpha_{i,j+1}^V$, hence IH5$_{i,j+1}$.

Let $q \in \Pi_{i,j+1}$. If $q \neq p$, the local state of q is the same in $C\left(\alpha_{i,j+1}^V\right)$ and in $C\left(\alpha_{i,j}^{\tilde{V}}\right)$, and by IH6$_{i,j}$, $C\left(\alpha_{i,j}^{\tilde{V}}\right)$

and $C\left(\alpha_{i,j}^{V_{V(q)}}\right)$ are indistinguishable to q, so $C\left(\alpha_{i,j+1}^{V}\right)$ and $C\left(\alpha_{i,j+1}^{V_{V(q)}}\right)$ are indistinguishable to q. Moreover, in all the steps of $\alpha_{i,j+1}^{V}$ following $\alpha_{i,j}^{\tilde{V}}$, p only read from (1) the state of instances of O, which are the same in $C\left(\alpha_{i,j}^{V_{V(p)}}\right)$ and $C\left(\alpha_{i,j}^{\tilde{V}}\right)$ by $\text{IH2}_{i,j}$, (2) the shared registers of X_i, that were overwritten by processes of Φ with the same value in $\alpha_{i,j+1}^{V}$ and $\alpha_{i,j+1}^{V_{V(p)}}$ by $\text{IH6}_{i,j}$ and by definition of $\tilde{V}$, and (3) the other shared registers that are in their initial state by $\text{IH5}_{i,j}$. Therefore, $C\left(\alpha_{i,j}^{V}\right)$ and $C\left(\alpha_{i,j}^{V_{V(p)}}\right)$ are indistinguishable to p. Hence, $\text{IH6}_{i,j+1}$.

From i to $i+1$ Suppose that, for some $i < m-1$, and for $j = (m-i)u_{i+1}$, we have built $\Pi_{i,j}$, $\mathcal{V}_{i,j}$ and $\alpha_{i,j}^{V}$ for all V, verifying the induction hypotheses $\text{IH1}_{i,j}$ to $\text{IH6}_{i,j}$ stated above.

We pose $\Pi_{i+1,0} = \Pi_{i,j}$, $\mathcal{V}_{i+1,0} = \mathcal{V}_{i,j}$ and for all valuations V of $\mathcal{V}_{i+1,0}$, $\alpha_{i+1,0}^{V} = \alpha_{i,j}^{V}$. By $\text{IH4}_{i,j}$ and the pigeon holes Theorem, there exists one of the $m-i$ registers in $X \setminus X_i$, that defines x_{i+1}, that is covered by at least u_{i+1} processes of $\Pi_{i+1,0}$.

Induction hypotheses $\text{IH1}_{i+1,0}$, $\text{IH2}_{i+1,0}$, $\text{IH5}_{i+1,0}$ and $\text{IH6}_{i+1,0}$ trivially follow from $\text{IH1}_{i,j}$, $\text{IH2}_{i,j}$, $\text{IH5}_{i,j}$ and $\text{IH6}_{i,j}$, respectively, and $\text{IH4}_{i+1,0}$ is a tautology when $j = 0$. By $\text{IH3}_{i,j}$, all registers of X_i are covered at least $u_i - j = u_{i+1}$ times in $C\left(\alpha_{i,j}^{V}\right)$, and so is x_{i+1} by construction, hence $\text{IH3}_{i+1,0}$.

Contradiction To conclude the proof, let us consider the last step in the induction above, with $i = m-1$ and $j = (m-i)u_{i+1} = 2$. By $\text{IH4}_{m-1,2}$, at least 2 processes cover shared objects in $X \setminus X_i = \emptyset$ by definition of m. This is absurd, so A cannot exist. □

Theorem 5 *Binary consensus has consensus number ∞_1^3.*

Proof As stated earlier, $M_1[\texttt{Cons}\langle\mathbb{B}\rangle]$ is universal, and, by Proposition 4, so is $MA_3[\texttt{Cons}\langle\mathbb{B}\rangle]$. Moreover, as $\texttt{Cons}\langle\mathbb{B}\rangle$ has a finite number of states, $M_2[\texttt{Cons}\langle\mathbb{B}\rangle]$ is not universal by Proposition 5. In conclusion, $\texttt{Cons}\langle\mathbb{B}\rangle$ has consensus number ∞_1^3. □

9 Objects with consensus number ∞_2^3

Because an object with consensus number ∞_1^3 is universal in MA_3 and an object with consensus number ∞_2^2 is universal in M_2, their composition can only have consensus number ∞_2^3 or ∞_3^3. In this section, we prove that the composition of binary consensus and iterator stacks, our respective examples for consensus numbers ∞_1^3 and ∞_2^2, is not universal in M_3 (Proposition 6), so it has consensus number ∞_2^3 (Theorem 6).

Similarly to propositions 3 and 5 , the proof of Proposition 6 proposes a scheduler that builds a Π'-partitioned execution, keeping track of a subset Π' of processes that have never communicated with each other, and in which more and more shared objects are covered (Definition 9 adapts the notion of coverage to take iterator stacks and binary consensus objects into account, and adds a property that applies to the whole configuration). The major difficulty is that iterator stacks cannot be overwritten by a finite number of processes, and the valency-based proof introduced in [2] cannot be adapted to a setting where binary consensus objects can be used in a critical configuration. Lemma 15 allows the scheduler to introduce a flow of newly arrived processes that, by covering, reading or writing all iterator stacks, prevents any chosen process trying to access an iterator stacks from learning any valuable information about the existence of other processes. This intuition is specified in Definition 10, by the concept of blind extensions.

Definition 9 (*Covered configuration*) An object x is write-covered by a process p in a configuration C if: (1) x is a register and the next step of p in C is a write on x, (2) x is a binary consensus object and the next step of p in C is to propose a value to x, or (3) x is an iterator stack and the next step of p in C is a write on x.

An object x is covered by a process p in a configuration C if x is write-covered, or x is an iterator stack and the next step of p in C is a read on x.

Let $\Pi' \subset \Pi$ be a set of processes, let $n \in \mathbb{N}$, and let Y be a set of shared objects. We say that a configuration C is (Π', n, Y)-covered if, in C, all objects in Y are covered at least n times by processes from Π', and, in the case of a binary consensus object x, at least n processes are about to propose the same value.

Definition 10 (*Blind extension*) Let α be a $(\Pi', \sim, p)$-partitioned execution. We say that $\alpha\beta$ is a blind extension of α if no process took steps in both α and β, and for each process q taking steps in β, there is an extension $\alpha_p\beta'$ of α_p such that the local state of q is the same in $C(\alpha\beta)$ and in $C(\alpha_p\beta')$. In other words, only fresh processes took steps in β, but they could not learn about the existence of processes other than those that are equivalent to p.

Lemma 15 *Let α be a $(\Pi', \sim, p)$-partitioned execution of a consensus algorithm A, let X be the set of objects instantiated at the set-up of A, and let $m = |X|$.*

For all $k \in \{0, \ldots, m\}$, there exists a blind extension $\alpha\beta$ of α such that either:

- *at least k different objects are write-covered in $C(\alpha\beta)$ by processes $q_1, \cdots, q_k$ that did not take steps in α, or*
- *some process q that did not take any step in α terminates its execution.*

Proof We prove the lemma by induction on k. For $k = 0$, we pose $\beta = \varepsilon$, the empty execution. Let us suppose, as the induction hypothesis $H(k)$, that the lemma holds for some $k \in \{0, \ldots, m-1\}$. We start the proof of $H(k+1)$ by proving a claim.

Claim There exists a blind extension $\alpha\beta$ of α such that either:

- at least k different objects are write-covered in $C(\alpha\beta)$ by processes $q_1, \ldots, q_k$ that did not take steps in α, and one more different object is covered in $C(\alpha\beta)$ by a process q_{k+1} that did not take steps in α, or
- some process q that did not take any step in α terminates its execution.

Proof Suppose this claim is false. We build an infinite execution $\alpha\beta_0\beta_1\beta_2\ldots$ in which some process q takes an infinite number of steps, such that each extension $\alpha\beta_0\ldots\beta_n$ of α is blind. Let w be the number of writes on iterator stacks in α, let $\alpha\beta_0 = \alpha\gamma_1\ldots\gamma_{w+2}$ be the blind execution obtained after invoking the induction hypothesis $H(k)$, $w+2$ times, and let Y_l be the set containing the k objects write-covered in γ_l, for each l. Let q be a process that did not take steps in $\alpha\beta_0$. As we supposed the claim was false, $Y = \bigcup_{l=1}^{w+2} Y_l$ has size k and each object $y \in Y$ is write-covered at least $w+2$ times in $C(\alpha\beta_0)$.

Suppose we have built a blind extension $\delta_n = \alpha\beta_0\ldots\beta_n$ of α. We build β_{n+1} as follows, such that $\alpha\beta_0\ldots\beta_n\beta_{n+1}$ is a blind extension of α. As we supposed the claim was false, q cannot terminate its execution in its next step.

- Suppose q is about to read an iterator stack $y \in Y$ in configuration $C(\delta_n)$. Let w' be the number of writes on some iterator stack in δ_n. We build $\delta_{n+1} = \delta_n\zeta_1\ldots\zeta_{w'}\eta$ as follows: each ζ_l is the result of one invocation of the induction hypothesis $H(k)$. As we supposed the claim was false, the set of write-covered objects in each ζ_l is Y. In particular, in $C(\delta_n\zeta_1\ldots\zeta_{w'})$, y is write-covered w' times by processes that did not take steps in δ_n. In η, we let w' processes write in y, then q reads in y and gets one of the values written by one of these processes, which ensures the extension is blind.
- If q is about to write into an iterator stack $y \in Y$ in configuration $C(\delta_n)$, β_{n+1} is solely composed of the next step of q. The write returns an iterator $i = i_\alpha + i'$, where i_α is the number of writes on y in α and i' is the number of writes on y in $\beta_0\ldots\beta_n$. As $i_\alpha \leq w$, q cannot distinguish the return value with a return value it would have had if its write in y was preceded by i_α writes from processes that arrived in β_0, so the extension is blind.
- Otherwise, in configuration $C(\delta_n)$, q is about to execute a local step, read from a register $x \in X$, write into a register $y \in Y$, propose a value to a consensus object $y \in Y$, or access an object instantiated during $\beta_0\ldots\beta_n$ or in α by some process $p' \sim p$. In all these cases, β_{n+1} is solely composed of the next step of q, which is a blind extension of δ_n.

Supposing the claim is false, we built an execution in which process q takes an infinite number of steps, which contradicts wait-freedom and concludes the proof of the claim. □

Let us continue the proof of Lemma 15 by supposing that $H(k+1)$ is false. We build an infinite execution $\alpha\beta_0\beta_1\beta_2\ldots$ in which some process q takes an infinite number of steps, and such that each extension $\alpha\beta_0\ldots\beta_n$ is blind.

Let w be the number of write operations on iterator stacks in α, and $w' = (m-k)(w^2+1)$. Remark that w' is an upper bound on the number of read operations that can return a non-$\perp$ value in $(m-k)$ iterator stacks, starting from $C(\alpha)$. We build $\alpha\beta_0 = \alpha\gamma_1\ldots\gamma_{w'+1}$ such that each γ_l is the blind extension given by the claim. As we supposed $H(k+1)$ was false, 1) a set Y of k objects are write-covered $(w'+1)$ times in $C(\alpha\beta_0)$ by processes that arrived in β_0, 2) no process wrote in an iterator stack $y \notin Y$ in β_0, and 3) $(w'+1)$ processes that did not take steps in α are about to read iterator stacks that are not in Y. Let Φ be the set of these $(w'+1)$ processes.

Let us suppose we have built a blind extension $\delta_n = \alpha\beta_0\ldots\beta_n$ of α such that some process in Φ took at least one step in β_l, for each $l \leq n$. To build β_{n+1}, we pick some process $q \in \Phi$ that did not read a value $v \neq \perp$ in an iterator stack $y \notin Y$ in δ_n. Such a process does exist because if the hypothesis that $H(k+1)$ is false then no process wrote in an iterator stack $y \notin Y$ in $\beta_0\ldots\beta_n$. On the other side, it is impossible to read a non-$\perp$ value in an iterator stack $y \notin Y$ more than $w' < |\Phi|$ times. Moreover, if the hypothesis that $H(k+1)$ is false, then q cannot terminate its execution in its next step.

- Suppose q is about to read from an iterator stack $y \in Y$ in configuration $C(\delta_n)$. Let w'' be the number of writes on iterator stacks in δ_n, and let us build $\delta_{n+1} = \delta_n\zeta_1\ldots\zeta_{2w''}\eta$ as follows. Let $\lambda_1 = \delta_n$ and $\lambda_i = \delta_n\zeta_1\ldots\zeta_{i-1}$ for $i > 1$. For each $i \in \{1, \ldots, 2w''\}$, $\lambda_i\zeta_i$ is the shortest blind extension of λ_i such that a process that did not take steps in λ_i, is about to write in y, or to read from y in the same iterator as q. Such an extension exists by the claim and the supposition that $H(k+1)$ is false. By the pigeon holes theorem, two cases are possible in $C(\delta_n\zeta_1\ldots\zeta_{2w''})$. If at least w'' new processes are about to read y, then η contains their read, and then the read by q returning $\perp$. Otherwise, at least w'' new processes are about to write in y, and η contains their write and the read by q, that returns a value written in η. In both cases, δ_{n+1} is blind.

- If q is about to write in an iterator stack $y \in Y$ in configuration $C(\delta_n)$, the only step of β_{n+1} is the write operation of q. As described in the claim, the fact that y is write-covered at least $(w'' + 1)$ times by processes arrived in β_0, which is more than the number of writes on y in α, implies that the extension is blind.
- In the other cases, in configuration $C(\delta_n)$, process q is about to execute a local step, to read from a register $x \in X$, to write into a register $y \in Y$, to propose a value to a consensus object $y \in Y$, or to access an object instantiated during $\beta_0 \dots \beta_n$ or in α by some process $q' \sim q$. In all these cases, β_{n+1} is solely composed of the next step of q, which is a blind extension of δ_n.

Assuming $H(k+1)$ is false, we have built an execution in which a finite number of processes takes an infinite number of steps, which contradicts wait-freedom and concludes the proof. □

Proposition 6 *$M_3[\texttt{Cons}\langle\mathbb{B}\rangle, \texttt{IStack}]$ is not universal.*

Proof Suppose there exists an algorithm A that solves consensus in $M_3[\texttt{Cons}\langle\mathbb{B}\rangle, \texttt{IStack}]$. Similarly to Proposition 3, we suppose that processes start the algorithm by writing their value to some register `first`, and finish it by writing their decided value to another register `last`. At the initialization of A, a finite set X of $|X| = m$ objects are created.

For all $i \in \{0, \dots, m\}$, we pose $u_i = (m-i+1)!2^{m-i+1}$. We consider an execution in which $\Pi = \{p_0, p_1, p_2, \dots\}$ is infinite and each process p_i proposes its identifier i to consensus. We will now build, by induction on $(i, j) \in \mathbb{N}^2$, with $0 \leq i < m$ and $0 \leq j \leq 2 \times (m-i) \times u_{i+1}$, taken in lexicographical order:

- A sequence $x_i \in X$ of shared objects of X (we define $X_i = \{x_0, \dots, x_i\}$);
- A sequence $\Pi_{i,j}$ of processes sets;
- A sequence $\alpha_{i,j}$ of $\Pi_{i,j}$-partitioned executions (let $\sim_{i,j}$ be the equivalence relation) leading to a $(\Pi_{i,j}, u_i - j, X_i)$-covered configuration, such that at least j processes of $\Pi_{i,j}$ cover objects that are not in X_i, and these objects are in their initial state.

Initialization for $i = 0$and $j = 0$ We pose $\Pi_{0,0} = \Pi$ and $x_0 = \texttt{first}$. In execution $\alpha_{0,0}$, each process $p_k \in \{p_0, \dots, p_{u_0}\}$ proposes its identifier k and stops executing when it is about to write in `first`. As no operation on shared objects has occured in $\alpha_{0,0}$, $\alpha_{0,0}$ is $\Pi_{0,0}$-partitioned and $C(\alpha_{0,0})$ is $(\Pi_{0,0}, u_0, X_0)$-covered.

From $j$to $j+1$,for a fixed i Suppose that, for some $i \in \{0, \dots, m-1\}$, we have built X_i, and for some $j < 2 \times (m-i) \times u_{i+1}$, we have built $\Pi_{i,j}$ and $\alpha_{i,j}$ verifying the properties stated above. We build $\alpha_{i,j+1} = \alpha_{i,j}\beta_{i,j}\gamma_{i,j}$ and $\Pi_{i,j+1}$ as follows.

Let us first remark that $\texttt{last} \notin X_i$. Otherwise, at least $u_i - j > 2$ processes $p_A \neq p_B \in \Pi_{i,j}$ would be about to write respectively a and b to the register `last` in $C\left(\alpha_{i,j}\right)$, such that a was proposed by process p_a and b was proposed by process p_b, with $p_a \sim_{i,j} p_A \nsim_{i,j} p_B \sim_{i,j} p_b$, by validity of consensus. Then, p_A and p_B would decide different values violating the agreement property of consensus.

Let us pick, arbitrarily, a set $\Phi \subset \Pi_{i,j}$ of $i+1$ processes, each covering a different register from X_i in $C\left(\alpha_{i,j}\right)$ (recall that each of them is covered at least $u_i - j > 1$ times). The extension $\beta_{i,j}$ is composed of one step of each process in Φ. Let us also pick, arbitrarily as well, a process $p \in \Phi$. We pose $\Pi' = (\Pi_{i,j} \setminus \Phi) \cup \{p\}$, and let $\sim'$ be the equivalence relation built by merging the classes of equivalence of processes of Φ, in $\sim_{i,j}$. Execution $\alpha_{i,j}\beta_{i,j}$ is $(\Pi', \sim', p)$-partitioned.

Let $\alpha_{i,j+1} = \alpha_{i,j}\beta_{i,j}\gamma_{i,j}$ be the shortest blind extension of $\alpha_{i,j}\beta_{i,j}$ such that, in $C(\alpha_{i,j}\beta_{i,j}\gamma_{i,j})$, some process q covers an object $y \notin X_i$ or terminate its execution. Such an extension exists by Lemma 15 for $k = i+1$. Since q cannot terminate itts execution before covering $\texttt{last} \notin X_i$, q covers an object $y \notin X_i$ in $C(\alpha_{i,j}\beta_{i,j}\gamma_{i,j})$. Moreover, as we considered the shortest such extension, objects that are not in X_i are still in their initial state.

Let $\Pi_{i,j+1} = (\Pi_{i,j} \setminus \Phi) \cup \{q\}$, and let $\sim_{i,j+1}$ be the equivalence relation built by merging the classes of equivalence of processes of Φ, in $\sim_{i,j}$ and adding all processes introduced in $\gamma_{i,j}$ in the class of equivalence of q. Execution $\alpha_{i,j+1}$ is $(\Pi_{i,j+1}, \sim_{i,j+1}, q)$-partitioned because $\alpha_{i,j}\beta_{i,j}$ is $(\Pi', \sim', p)$-partitioned and $\alpha_{i,j+1}$ is a blind extension of $\alpha_{i,j}\beta_{i,j}$. Moreover, $C\left(\alpha_{i,j+1}\right)$ is $(\Pi_{i,j+1}, u_i - (j+1), X_i)$-covered by the same processes as $C\left(\alpha_{i,j}\beta_{i,j}\right)$, and q, as well as j processes from $\Pi_{i,j}$, cover objects that are not in X_i.

From $i$to $i+1$ Suppose that, for some $i \in \{0, \dots, m-2\}$ and $j = 2 \times (m-i) \times u_{i+1}$, we have built X_i, $\Pi_{i,j}$ and $\alpha_{i,j}$ verifying the properties stated above.

At least $j = 2 \times (m-i) \times u_{i+1}$ objects are covered in $\alpha_{i,j}$. By the pigeon holes theorem, there exists one of the $m-k$ objects that is not in X_i, denoted by x_{i+1}, covered by at least $2 \times u_{i+1}$ processes. If x_{i+1} is a binary consensus object, by the pigeon holes theorem again, the most proposed value is proposed at least u_{i+1} times. Let us denote by Φ the set of these processes. Moreover, at least $u_i - j = 2u_{i+1} > u_{i+1}$ processes of $\Pi_{i,j}$ cover each object in X_i. Let us denote by Ψ the set of these processes.

We pose $\alpha_{i+1,0} = \alpha_{i,j}$ and $\Pi_{i+1,0} = \Phi \cup \Psi$. Execution $\alpha_{i+1,0}$ is $\Pi_{i+1,0}$-partitioned and $C(\alpha_{i+1,0})$ is $(\Pi_{i+1,0}, u_{i+1}, X_{i+1})$-covered, which concludes the induction.

Contradiction Let us consider the last step in the induction above, with $i = m-1$ and $j = 2 \times (m-i) \times u_{i+1} = 4$.

At least 4 processes cover shared objects in $X \setminus X_i = \emptyset$ by definition of m. This is absurd, so A cannot exist. □

Theorem 6 *The composition of iterator stacks and binary consensus has consensus number* ∞_2^3.

Proof By [2], M_2[IStack] is universal, and by Proposition 4, MA_3[Cons⟨$\mathbb{B}$⟩] is universal, so IStack + Cons⟨$\mathbb{B}$⟩ has at least consensus number ∞_2^3. By Proposition 6, IStack + Cons⟨$\mathbb{B}$⟩ has at most consensus number ∞_2^3. □

10 Conclusion

This paper explores the universality of shared objects in the infinite arrival model where it is not possible to allocate and initialize, at once, an infinite number of memory locations. For that, we extend the existing wait-free hierarchy by separating the objects having an infinite consensus number into five categories, according to their universality in the bounded, finite or infinite arrival models, and the need or not of an infinite memory allocation mechanism. This paper raises several new open issues, that we detail thereafter.

We proposed a universal construction using consensus objects and read/write registers, in which all invoked operations are stored twice in infinitely growing logs. Although this construction serves the purpose of proving the universality of consensus in all considered models, its complexity makes it impractical. An interesting open problem is the space complexity of universal constructions in multi-threaded systems, including in situations where different special instructions, such as compare-and-swap, are available.

We supposed that processes share an infinite memory. Although this assumption is central to the definition of the Turing Machine at the basis of computer science, it naturally implies that pointers to memory locations have infinite size, which is less practical. Without this assumption, multi-valued consensus could be solved using a number of binary consensus objects equal to the size of a pointer [23]. An interesting open problem is the existence of a shared object with consensus number ∞_1^3 that does not have a poly-logarithmic implementation of consensus in MA_2.

Finally, the example of an object having consensus number ∞_2^3 we exhibited in this paper is a composition of two objects having a consensus number resp. ∞_2^2 and ∞_1^3. It would be interesting to investigate if this is always the case. This can be split into two questions. Does there exist an object of consensus number ∞_2^3 that cannot be expressed as such a composition? Conversely, does there exist two objects of consensus number ∞_2^2 and ∞_1^3 whose composition has consensus number ∞_3^3?

Declarations

Conflict of interest The authors declare that they have no conflict of interest.

References

1. Afek, Y., Gafni, E., Morrison, A.: Common2 extended to stacks and unbounded concurrency. Distrib. Comput. **20**(4), 239–252 (2007)
2. Afek, Y., Morrison, A., Wertheim, G.: From bounded to unbounded concurrency objects and back. In: Proceedings of the 30th Annual ACM Symposium on Principles of Distributed Computing, PODC 2011, San Jose, CA, USA, pp. 119–128 (2011)
3. Afek, Y., Weisberger, E., Weisman, H.: A completeness theorem for a class of synchronization objects. In: Proceedings of the Twelfth Annual ACM Symposium on Principles of Distributed Computing, pp. 159–170 (1993)
4. Aguilera, M.K.: A pleasant stroll through the land of infinitely many creatures. ACM SIGACT News **35**(2), 36–59 (2004)
5. Attiya, H., Bar-Noy, A., Dolev, D., Peleg, D., Reischuk, R.: Renaming in an asynchronous environment. J. ACM **37**(3), 524–548 (1990)
6. Censor-Hillel, K., Petrank, E., Timnat, S.: Help!. In: Proceedings of the 2015 ACM Symposium on Principles of Distributed Computing, pp. 241–250. ACM (2015)
7. Ellen, F., Gelashvili, R., Shavit, N., Zhu, L.: A complexity-based hierarchy for multiprocessor synchronization. In: Proceedings of the 2016 ACM Symposium on Principles of Distributed Computing, pp. 289–298 (2016)
8. Fatourou, P., Kallimanis, N.D.: Highly-efficient wait-free synchronization. Theory Comput. Syst. **55**(3), 475–520 (2014)
9. Filipović, I., O'Hearn, P., Rinetzky, N., Yang, H.: Abstraction for concurrent objects. Theoret. Comput. Sci. **411**(51–52), 4379–4398 (2010)
10. Fischer, M.J., Lynch, N.A., Paterson, M.: Impossibility of distributed consensus with one faulty process. J. ACM **32**(2), 374–382 (1985)
11. Gafni, E., Merritt, M., Taubenfeld, G.: The concurrency hierarchy, and algorithms for unbounded concurrency. In: Proceedings of the Twentieth Annual ACM Symposium on Principles of Distributed Computing, pp. 161–169. ACM (2001)
12. Herlihy, M.: Wait-free synchronization. ACM Trans. Program. Lang. Syst. **13**(1), 124–149 (1991)
13. Herlihy, M., Rajsbaum, S., Raynal, M.: Power and limits of distributed computing shared memory models. Theor. Comput. Sci. **509**, 3–24 (2013)
14. Herlihy, M., Shavit, N.: The Art of Multiprocessor Programming. Morgan Kaufmann, Shavit (2008)
15. Herlihy, M., Wing, J.M.: Linearizability: a correctness condition for concurrent objects. ACM Trans. Program. Lang. Syst. **12**(3), 463–492 (1990)
16. Lamport, L.: A new solution of Dijkstra's concurrent programming problem. Commun. ACM **17**(8), 453–455 (1974)
17. Mostefaoui, A., Raynal, M., Tronel, F.: From binary consensus to multivalued consensus in asynchronous message-passing systems. Inf. Process. Lett. **73**(5–6), 207–212 (2000)
18. Plotkin, S.A.: Sticky bits and universality of consensus. In: Proceedings of the Eighth Annual ACM Symposium on Principles of Distributed Computing, pp. 159–175 (1989)
19. Raynal, M.: Concurrent Programming - Algorithms, Principles, and Foundations. Springer (2012)
20. Raynal, M.: Distributed universal constructions: a guided tour. Bull. EATCS, **121**, 2017

21. Taubenfeld, G.: Distributed Computing Pearls. Synthesis Lectures on Distributed Computing Theory. Morgan & Claypool Publishers (2018)
22. Treiber, R.K.: Systems Programming: Coping with Parallelism. International Business Machines Incorporated, Thomas J. Watson Research (1986)
23. Zhang, J., Chen, W.: Bounded cost algorithms for multivalued consensus using binary consensus instances. Inf. Process. Lett. **109**(17), 1005–1009 (2009)