Extending the Wait-free Hierarchy to Multi-Threaded Systems

Matthieu Perrin, Achour Mostefaoui, Grégoire Bonin

Extending the Wait-free Hierarchy to Multi-Threaded Systems

Matthieu Perrin
LS2N, Université de Nantes
matthieu.perrin@univ-nantes.fr

Achour Mostéfaoui
LS2N, Université de Nantes
achour.mostefaoui@univ-nantes.fr

Grégoire Bonin
LS2N, Université de Nantes
gregoire.bonin@univ-nantes.fr

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: 1) new threads can be created dynamically at runtime, so there is no bound on the number of threads participating in a long-running execution. 2) threads have access to a memory allocation mechanism that cannot allocate infinite arrays. This makes it challenging to adapt some algorithms to multi-threaded systems, especially those that assign one shared register per process.

This paper explores the synchronization power of shared objects in multi-threaded systems by extending the famous wait-free hierarchy to take these constraints into consideration. It proposes to subdivide the set of objects with an infinite consensus number into five 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. It then exhibits one object illustrating each proposed degree.

CCS CONCEPTS

• **Theory of computation → Distributed computing models**; • **Software and its engineering → Process synchronization**; • **Computer systems organization → Multicore architectures**; **Dependable and fault-tolerant systems and networks**.

KEYWORDS

Arrival models, Consensus number, Linearizability, Memory allocation, Multi-Threaded System, Universality, Wait-freedom

ACM Reference Format:
Matthieu Perrin, Achour Mostéfaoui, and Grégoire Bonin. 2020. Extending the Wait-free Hierarchy to Multi-Threaded Systems. In *ACM Symposium on Principles of Distributed Computing (PODC '20), August 3–7, 2020, Virtual Event, Italy*. ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3382734.3405723

1 INTRODUCTION

Wait-free universality. In sequential computing, the notion of universality is represented by a Turing machine that can compute all 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 [Fischer et al. 1985]. 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 [Herlihy et al. 2013].

A distributed system can be abstracted as a set of processes accessing concurrently a set of concurrent 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 [Herlihy and Shavit 2008; Raynal 2012; Taubenfeld 2018]. 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 [Herlihy and Wing 1990] is a consistency condition. It ensures that all the operations of a distributed history appear as if they were executed sequentially: each operation appears 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.

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 [Herlihy 1991] and lock-freedom [Herlihy and Wing 1990]. 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 classical asynchronous distributed systems composed of n processes among which all but one may crash, the consensus problem has no deterministic solution using only basic read/write registers [Loui and Abu-Amara 1987]. 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 [Herlihy 1991], 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.

PODC '20, August 3–7, 2020, Virtual Event, Italy

ACM ISBN 978-1-4503-7582-5/20/08…$15.00
https://doi.org/10.1145/3382734.3405723

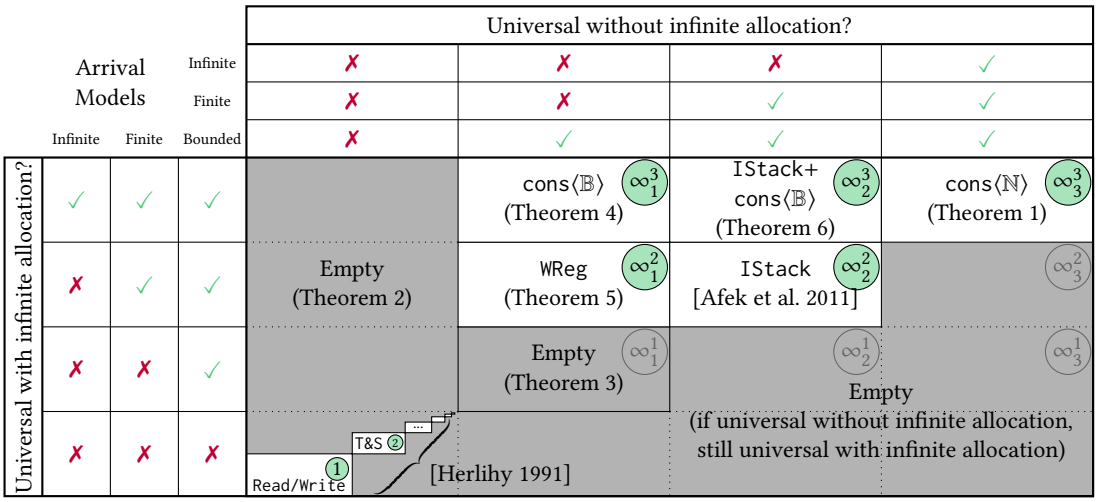

Figure 1: Extended wait-free hierarchy: in a multi-treaded system, it is impossible to implement an object O_1 using any number of instances of O_2 and read/write registers, if O_2 is more on the left, or bottom, than O_1. Green circles display the consensus number of each degree.

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 common memory space in which they can allocate a (virtually) unbounded but finite number of memory locations to instantiate record data structures or finite arrays. In particular, when no bound is known on the number of threads in an execution, assigning one register to each of them is not trivial. This fact is often regarded as secondary when designing concurrent algorithms. For example, [Aguilera 2004] identifies as "trivial" the change of finite arrays indexed by processes to infinite arrays or linked lists. A consequence of this paper is that maintaining extensible data structures such as linked lists requires a synchronization power that is not necessarily provided by all objects that 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 [Afek et al. 2011], 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 over time an execution. The present paper answers the following question: how to compare the synchronization power of shared objects in multi-threaded systems?

Approach. Following the same approach as in [Herlihy 1991], 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 [Gafni et al. 2001]. In these models, any number of processes can 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. Four arrival models are distinguished in [Aguilera 2004]: [1]

- The classical model, M_1^n where the number n of processes is fixed and may appear in the process code.
- 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, 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.

Moreover, the impossibility to allocate infinite arrays is an important limiting factor that restricts the computing power of some objects in multi-threaded systems. We also study the synchronization power of shared objects according to the possibility, or not, to

[1] A fifth model, M_4, called *infinite concurrency*, was introduced in [Aguilera 2004], 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.

allocate infinite arrays. Hence, we propose the two-dimensional hierarchy presented on Figure 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.

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 Figure 1. We prove that multi-valued consensus (denoted $\mathsf{cons}\langle\mathbb{N}\rangle$) 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 [Afek et al. 2011], we naturally deduce that iterator stacks have consensus number ∞_2^2. Interestingly, we prove that binary consensus (denoted $\mathsf{cons}\langle\mathbb{B}\rangle$) is not universal in multi-threaded systems, resulting in a consensus number of ∞_1^3 (Theorem 4). 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 show that a window register (denoted WReg), in which a read operation returns the k last written values (k chosen at initialization), has consensus number ∞_1^2 (Theorem 5).

Organization of the paper. The remainder of this paper is organized as follows. We first present the infinite arrival models in Section 2. Then, Section 3 shows that consensus is still universal in the infinite arrival model. Sections 4 and 5 identify the empty degrees of the hierarchy by proving theorems 2 and 3. Sections 6, 7 and 8 show that the remaining degrees are not empty by providng the consensus number of binary consensus, window registers and a composition of binary consensus and iterator stacks. Finally, Section 9 concludes the paper.

2 COMPUTING MODELS

This paper considers distributed computations where processes (or threads) have access to local memory for local computations and have also access to shared objects (shared memory) to communicate and synchronize. 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 instantiate 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.

2.1 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:

- **Classical model M_1^n:** $|\Pi| = n$, and n is a parameter of the system model.
- **Bounded arrival model M_1:** $|\Pi|$ is finite and known by the processes at runtime.
- **Finite arrival model M_2:** $|\Pi|$ is finite but unknown to the processes.
- **Infinite arrival model M_3:** $|\Pi|$ is countable.

2.2 Communication between processes

Processes communicate by reading and writing a memory composed of an infinite number of unbounded registers [2]. Processes have access to an allocation mechanism that can only return an unbounded, but finite, number of memory locations at once and that never allocates twice the same memory location.

Processes are not limited in the number of registers they can access during an execution. However, they can only access memory locations allocated either at the system set up, or returned by the allocation mechanism, or by following references stored (as integer values) in some accessible memory location.

As advocated in the Introduction, when strong enough synchronization objects are not available, it may be necessary to enable the allocation mechanism 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.

2.3 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 square brackets in the model name and referred to as enriching shared objects. For example, $M_3[\mathsf{cons}\langle\mathbb{N}\rangle]$ denotes the infinite arrival model where as many consensus objects as necessary are made available. Each enriching shared object is atomic in the sense that the different executions of the calls to its operations are totally ordered.

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

Window registers. A window register of size k [Perrin et al. 2016] (denoted k-WReg) has two operations: a write operation write(v), that takes an integer argument and returns no value, and a read operation read() that returns the ordered list of the last k values written. Some default values $\perp$ may appear, if less than k writes precede the read. Note that the k-WReg generalizes the classical read/write register that corresponds to the special case $k = 1$. WReg is the generalization in which window registers of any sizes are accessible: the size k is passed as an argument of the constructor.

Iterator stacks. The iterator stack IStack, introduced in [Afek et al. 2011], 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 k^{th} invocation of isRead(i) returns the k^{th} value ever written if isWrite was invoked at least $\max(i + 1, k)$ times, and $\perp$ otherwise.

Consensus objects. A consensus object, denoted cons$\langle T \rangle$, provides one operation propose(v) that 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. We distinguish between binary consensus cons$\langle \mathbb{B} \rangle$ in which only values **true** and **false** can be proposed, and the multi-valued consensus, for example cons$\langle \mathbb{N} \rangle$ in which proposed values are integer values. Finally, cons$^n\langle T \rangle$ designates n-processes consensus, in which the previously-stated properties are only verified by the first n invocations of propose, and the next returned values are left unspecified.

2.4 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) [Filipović et al. 2010].

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.*

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

3 UNIVERSALITY OF CONSENSUS IN M_3

In order to prove the universality of consensus in the bounded arrival model, Herlihy introduced the notion of universal construction[3]. It is a generic algorithm that, given a sequential specification of any object whose operations are total[4], provides a concurrent implementation of this object. Wait-free implementations rely on what is called a helping mechanism recently formalized in [Censor-Hillel et al. 2015]. 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.

Algorithm 1 presents a universal construction close to the one presented in [Herlihy and Shavit 2008], except that the array of single-writer/multiple-reader registers used by processes to announce their operations is replaced by a weak log data structure. This data structure has been briefly described in [Bonin et al. 2019].

[3] A small guided tour on universal constructions can be found in [Raynal 2017].
[4] This means that any operation on the object can be called and the call returns regardless of the state of the object.

It allows processes to append a value and get back 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 successive returned sequences. However, any value appended by a correct process will eventually appear in all returned sequences (Lemma 3) and the order of the values of a returned sequence is never contradicted by subsequent sequences (Lemma 2).

A universal construction emulates any shared object. Which object is represented by an initial state initialState and a set of operations that change the state and return a value. Processes executing Algorithm 1 share three objects: operations, first and last.

- operations is a consensus object that gives access to a linked list of operations, called *strong log*, whose order defines the linearization order of the operations. This list is composed of nodes of the form $\langle v, \text{cons}\rangle$, where v is the invocation of a process and cons a consensus object that, when won by some process, will reference the next node of the list.
- first is a consensus object that references the head of a list of lists of appended values, called *weak log*. It accepts values of the form list = ⟨list.head, list.tail⟩ to be proposed, where list.tail is a consensus object that stores values of the same type as first, representing the next node in the list of lists. list.head is a node of the side list of the form node = ⟨node.head, node.tail⟩, where node.head is a value appended by some process and node.tail is a consensus object containing values of the same type as node.
- last is a read/write register referencing a consensus object of the same type as first, and initialized to the address of first. In absence of concurrency, last references the consensus object at the tail of the last list of the weak log.

When a process p_i invokes append(invoc), it first inserts its invocation invoc to the weak log (lines 1 to 3), then it reads the weak log to obtain the list of all previous and concurrent invocations that require helping (lines 4 to 8), and finally inserts the invocations it needs to help, including invoc, to the strong log (lines 9 to 14).

The main difficulty in the implementation of a weak log lies in the allocation of one memory location per process, where it can safely announce its invoked operation. Algorithm 1 solves this issue by using a novel feature, that we call *passive helping*: when a process wins a consensus (Line 1), it creates a side list to host invocations 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 (Line 3). Note that the consensus and the write on lines 1 and 2 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 1, is that last eventually moves forward, allowing very slow processes to get a place in a side list.

Definition 3 formalizes the order in which values are ordered in the weak log. Intuitively, this order is the concatenation of all the side lists. In lines 4 to 8, p_i traverses the weak log in this precedence order to build the sequence toHelp_i ($\oplus$ represents concatenation), which ensures consistency between the helping sequences of all processes (Lemma 2).

operation apply(invoc) **is**

1 $\quad \text{list}_i \leftarrow \text{last.read}().\text{propose}(\langle\langle \text{invoc}, \bot\rangle, \bot\rangle)$;
2 $\quad \text{last.write}(\text{list}_i.\text{tail})$; $\text{node}_i \leftarrow \text{list}_i.\text{head}$;
3 $\quad$ **while** $\text{node}_i.\text{head} \neq \text{invoc}$ **do**
$\qquad \text{node}_i \leftarrow \text{node}_i.\text{tail.propose}(\langle \text{invoc}, \bot\rangle)$;
4 $\quad \text{list}_i \leftarrow \text{first}$; $\text{node}_i \leftarrow \text{list}_i.\text{head}$;
$\qquad \text{toHelp}_i \leftarrow [\text{node}_i.\text{head}]$;
5 $\quad$ **while** $\text{node}_i.\text{head} \neq \text{invoc}$ **do**
6 $\qquad$ **if** $\text{node}_i.\text{tail} \neq \bot$ **then** $\text{node}_i \leftarrow \text{node}_i.\text{tail}$;
7 $\qquad$ **else** $\text{list}_i \leftarrow \text{list}_i.\text{tail}$; $\text{node}_i \leftarrow \text{list}_i.\text{head}$;
8 $\qquad \text{toHelp}_i \leftarrow \text{toHelp}_i \oplus [\text{node}_i.\text{head}]$;
9 $\quad \text{cons}_i \leftarrow \text{operations}$; $\text{state}_i \leftarrow \text{initialState}$;
10 $\quad$ **while** $\text{toHelp}_i \neq \varepsilon$ **do**
11 $\qquad \langle \text{winner}_i, \text{cons}_i\rangle \leftarrow \text{cons}_i.\text{propose}(\langle \text{toHelp}_i[0], \bot\rangle)$;
12 $\qquad \text{toHelp}_i \leftarrow \text{toHelp}_i \setminus \text{winner}_i$;
13 $\qquad$ **if** $\text{winner}_i = \text{invoc}$ **then**
$\qquad\qquad \text{result}_i \leftarrow \text{state}_i.\text{invoke}(\text{winner}_i)$;
14 $\qquad$ **else** $\text{state}_i.\text{invoke}(\text{winner}_i)$;
15 $\quad$ **return** result_i;

Algorithm 1: Universal construction in Model $M_3[\text{cons}\langle\mathbb{N}\rangle]$

Definition 3 (Weak log precedence). *A list* list *precedes a list* list′ *in the weak log if there exists a sequence of lists* $\text{list}_1, \ldots, \text{list}_n$ *such that* $\text{list} = \text{list}_1$, $\text{list}' = \text{list}_n$, *and for all* $1 \leq k < n$, $\text{list}_k.\text{tail} = \text{list}_{k+1}$. *A node* node *precedes a node* node′ *in the weak log if there exists a sequence of nodes* $\text{node}_1, \ldots, \text{node}_n$ *such that* $\text{node} = \text{node}_1$, $\text{node}' = \text{list}_n$, *and for all* $1 \leq k < n$, $\text{node}_k.\text{tail} = \text{node}_{k+1}$, *or if there exists a list* list *that precedes a list* list′, *such that* list.head *precedes* node *and* list′.head *precedes* node′. *A value* v *precedes a value* v *in the weak log if if there exists a node* node *that precedes a node* node′, *such that* node.head $= v$ *and* node′.head $= v'$.

Finally, p_i tries to insert all invocations of toHelp_i at the end of the list operations, in the order it read them. 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 [Herlihy and Shavit 2008], so the proof of Proposition 1 is similar. The proof of wait-freedom (Proposition 2) is more challenging because the proof of [Herlihy and Shavit 2008] heavily relies on the fact that the number of processes is finite.

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

Lemma 1 (Progress on last). *If an infinite number of processes execute Line 2, then the number of processes that read the same* last *value at Line 1 is finite.*

Lemma 2 (Total order of the weak log). *If two processes p_i and p_j terminate their invocations, then all pairs of values that both w_i and w_j contain appear in the same order.*

Lemma 3 (Eventual visibility in the weak log). *If some process p_i terminates its invocation, then the number of returned sequences that do not contain v_i is finite.*

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

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

Remark 1. *The usual algorithm to solve 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.*

4 INFINITE MEMORY ALLOCATION IS NOT NECESSARY IN M_1

The original paper on the wait-free hierarchy [Herlihy 1991] 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 coincides with Herlihy's one for objects with a finite consensus number, which justifies our choice to keep the same name.

This results 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 4). Such a bound can be obtained by using renaming algorithms. For example, [Attiya et al. 1990] do 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 4. *For any object O, if $\mathsf{cons}\langle\mathbb{B}\rangle$ can be implemented in $MA_1^n[O]$, then $\mathsf{cons}\langle\mathbb{B}\rangle$ can be implemented in $M_1^n[O]$.*

Proof. Suppose there exists an algorithm A that implements binary consensus in $MA_1^n[O]$. 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. We prove the following claim:

Claim 1. *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.*

Proof. Let us suppose this is not the case. We build, recursively, an infinite execution in which an infinite number of configurations are accessible at each step. Initially, let $\alpha_0 = \varepsilon$, the execution containing no step. Suppose we have built an execution α_k containing k steps, such that an infinite number of configurations are reachable from $C(\alpha_k)$. From $C(\alpha_k)$, a step corresponds to the next step of one of the processes that has not decided yet, so there are at most n different possible steps. At least one of them, β_k, leads to a configuration from which an infinite number of configurations are reachable. Let us pose $\alpha_{k+1} = \alpha_k\beta_k$.

As $(\alpha_k)_{k\in\mathbb{N}}$ is built such that α_k is a prefix of α_{k+1}, it converges to an infinite execution $\alpha = \beta_0\beta_1\beta_2\ldots$ Some process takes an infinite number of steps in α, so A is not wait-free. This is a contradiction. □

The number of possible inputs is bounded by $2^N \times 2^n$. For each of them, the number of accessible configurations is finite by Claim 1. Therefore, 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, $\mathsf{cons}\langle\mathbb{B}\rangle$ can be implemented in $MA_1^n[O]$. By Lemma 4, $\mathsf{cons}\langle\mathbb{B}\rangle$ can be implemented in $M_1^n[O]$. It is possible to implement $\mathsf{cons}\langle\mathbb{N}\rangle$ using a bounded number of $\mathsf{cons}\langle\mathbb{B}\rangle$ objects in the bounded arrival model using an algorithm like the one given in [Zhang and Chen 2009] and that can be easily adapted to shared memory [Raynal 2012]. Finally, by Theorem 1, O is universal in $M_1^n[O]$. □

Corollary 1. *For any object O, if $MA_1[O]$ is universal, then $M_1[O]$ is also universal.*

5 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 2 then uses these $\mathsf{cons}^n\langle\mathbb{N}\rangle$ objects to solve Consensus in MA_2 (Lemmas 5, 6 and 7).

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 $\mathsf{cons}^n\langle\mathbb{N}\rangle$ object that accepts participation of processes $p_0, \ldots, p_{r-1}$; and adopt$[r]$ is a register, initially $\perp$, 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 2 is round-based. At round r, processes with identifiers smaller than r agree on some value using the $\mathsf{cons}^n\langle\mathbb{N}\rangle$ 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.

Claim 2. *For any round r, at most r processes invoke* propose *on* cons$[r]$ *Line 4.*

Lemma 5 (Wait-freedom). *All executions of Algorithm 2 terminate in MA_2.*

Lemma 6 (Agreement). *If processes p_i and p_j decide respectively v_i and v_j, then $v_i = v_j$.*

Lemma 7 (Validity). *If p_i decides v, then a process proposed v.*

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

operation propose(val_i) **is**
1 $v_i \leftarrow val_i$;
2 **for** $r_i = 0, 1, 2, \ldots$ **do**
3 **if** $i \geq r_i$ **then** greaterId[r_i] $\leftarrow$ **true** ;
4 **else** adopt[r_i] $\leftarrow$ cons[r_i].propose(v_i) ;
5 **if** $\neg$greaterId[r_i] **then return** adopt[r_i] ;
6 **if** adopt[r_i] $\neq \perp$ **then** $v_i \leftarrow$ adopt[r_i] ;

Algorithm 2: Consensus in Model $MA_2[\mathsf{cons}^n\langle\mathbb{N}\rangle]$ (code for p_i)

6 BOOLEAN CONSENSUS HAS CONSENSUS NUMBER ∞_1^3

An object has consensus number ∞_1^3, if an infinite memory allocation is necessary to make it universal in the finite arrival model, and sufficient in the infinite arrival model. One reason why this could happen is because the number of instances necessary for synchronization grows boundlessly with the number of processes. Recently, [Ellen et al. 2016] introduced a complexity-based hierarchy ranking shared objects according to the number of instances necessary to solve obstruction-free consensus. For example, at least $\lceil \frac{n-1}{k} \rceil$ k-window registers are necessary to solve multi-valued consensus. Similarly, to our knowledge, no known algorithm uses less than $\log_2(n)$ binary consensus objects to solve consensus [Zhang and Chen 2009] in the worst case. In order to be universal in M_2, any object has only two ways to circumvent the limitation that only a fixed and finite number of objects can be created at initialization of any algorithm: either it has a constant complexity in Ellen et al's hierarchy, 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. In this section, we prove that $\mathsf{cons}\langle\mathbb{B}\rangle$ is universal in MA_3 (Proposition 3) but not in M_2 (Proposition 4), i.e. binary consensus has consensus number ∞_1^3 (Theorem 4).

The sticky bit object, a resettable version of binary consensus, has been shown to be universal in MA_1 in [Plotkin 1989]. Reductions of multi-valued consensus to binary consensus have later been proposed for message-passing systems [Mostefaoui et al. 2000], and extended to M_1 [Raynal 2012]. Algorithm 3 extends this result to the model MA_3. Processes share three infinite arrays propose, isSet and cons: for each index $j \in \mathbb{N}$, propose[j] is intended to store the value proposed by p_j, 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 proposed in the increasing order of the identifiers, until it agrees with other participants on a value proposed[j] that can be decided.

Proposition 3. *$MA_3[\mathsf{cons}\langle\mathbb{B}\rangle]$ is universal.*

Proposition 4 below shows that neither binary consensus nor window registers are universal in the finite arrival model when infinite memory allocation is impossible. The proof has the same flavor as the proofs in [Ellen et al. 2016], but simplified as we are only interested in decidability whereas their bounds need to be tight. More precisely, the proof of Proposition 4 builds a scheduler that keeps

operation propose(val_i) **is**
1 proposed[i] $\leftarrow val_i$; isSet[i] $\leftarrow$ **true**;
2 **for** $j = 0, 1, 2, \ldots$ **do**
3 **if** cons[j].propose(isSet[j]) **then return** proposed[j];

Algorithm 3: Consensus in $MA_3[\mathsf{cons}\langle\mathbb{B}\rangle]$ (code for p_i)

track of a subset Π' of processes that have never communicated with each other because they always propose the same values in binary consensus objects, and the values they write in window registers are overwritten. The property maintained by the executions produced by this scheduler, called Π'-partitioning, is specified in Definition 4. The scheduler builds an execution in which a large number of processes participate, and more and more shared objects are covered by many processes (i.e. these processes try to write in the objects, see. Definition 5) that do not know the existence of each other, until all objects are covered and two processes decide different values.

Definition 4 (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 and consensus objects have the same value in $C(\alpha)$ and $C(\alpha_p)$.*

Definition 5 (Covered configuration). *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, for every object $x \in Y$, 1) x is a window register and the next step of at least n processes from Π' is a write on x, or 2) x is a binary consensus object and there exists a common value $v \in \mathbb{B}$ such that next step of at least n processes from Π' is to propose v on x.*

Let p be a process and x be an object, we say that p covers x in a configuration C if C is $(\{p\}, 1, \{x\})$-covered, i.e. if p is about to write to x or to propose some value to x.

Proposition 4. *$M_2[\mathsf{cons}\langle\mathbb{B}\rangle, \mathtt{WReg}]$ is not universal.*

Proof. Let us suppose there exists an algorithm A that solves consensus in $M_2[\mathsf{cons}\langle\mathbb{B}\rangle, \mathtt{WReg}]$. 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 in any consensus algorithm without loss of generality. At the initialization of A, a finite set X of $m = |X|$ objects are created. We identify read/write registers and 1-WReg objects in this proof. Let l be a positive integer greater than the size of all window registers in X, and let $u_i = (m - i + 1)! \times (2l)^{(m-i+1)}$, for all $i \in \{0, ..., m\}$. We consider executions of processes in the finite set $\Pi = \{p_1, \ldots, p_{u_1}\}$.

We build, inductively, a sequence $(\Pi_i)_{1 \leq i \leq m}$ of process sets, a sequence $(x_i)_{1 \leq i \leq m}$ of shared objects of X (we note $X_i = \{x_1, \ldots, x_i\}$) and a growing sequence $(\alpha_i)_{1 \leq i \leq m}$ of Π_i-separable executions leading to a (Π_i, u_i, X_i)-covered configuration.

In execution α_1, each process $p_i \in \Pi$ proposes its own identifier i and stops executing when it is about to write in x_1 = first. We

define $\Pi_1 = \Pi$ and $\sim_1$ as the equality of processes. Execution α_1 is Π_1-separable because no process accessed any shared object, and $C(\alpha_1)$ is (Π_1, u_1, X_1)-covered by construction.

Suppose we have built a process set Π_i, a sequence of objects $x_1, \ldots, x_i$, and a Π_i-separable execution α_i leading to a (Π_i, u_i, X_i)-covered configuration, for some $i \in \{0, \ldots, m-1\}$. Let $v = 2 \times (m-i) \times u_{i+1}$. We build, inductively, a sequence of executions $\alpha_i^0, \ldots, \alpha_i^v$ and a sequence of process sets $\Pi_i^0, \ldots, \Pi_i^v$, such that for all $j \in \{0, \ldots, v\}$, α_i^j is $(\Pi_i^j, \sim_i^j, p)$-separable (for some $\sim_i^j$ and p), $C(\alpha_i^j)$ is $(\Pi_i^j, (u_i - j), X_i)$-covered, and j processes of Π_i^j cover objects that are not in X_i. Initially, let us pose $\alpha_i^0 = \alpha_i$ and $\Pi_i^0 = \Pi_i$. Suppose we have built α_i^j and Π_i^j for some $j < v$. We build $\alpha_i^{j+1} = \alpha_i^j \beta_i^j \gamma_i^j$ as follows.

Let us pick, arbitrarily, a set Φ of $l \times i$ processes from Π_i^j, such that $C(\alpha_i^j)$ is (Φ, l, X_i)-covered. Φ exists because $C(\alpha_i^j)$ is $(\Pi_i^j, (u_i - j), X_i)$-covered, and $u_i - j > l$. The extension β_i^j is composed of one step of each process $q \in \Phi$. Let us pick arbitrarily a process $p \in \Phi$, and let $\Pi' = (\Pi_i^j \setminus \Phi) \cup \{p\}$, and $\sim'$ be the equivalence relation built by merging the classes of equivalence of processes in Φ in $\sim_i^j$. As l is greater than the size of all window registers, those were overwritten in β_i^j by values written by processes in Φ, so $\alpha_i^j \beta_i^j$ is $(\Pi', \sim', p)$-partitioned.

Let us build γ_i^j by executing p until it covers an object that is not in $\{x_1, \ldots x_i\}$. Such a situation must happen because, 1) as A is wait-free, p cannot run in isolation forever, and 2) if p decides some value v proposed by some process $p_v \sim_{i+1} p$, then $l \times i$ other processes $p' \nsim_i p_v$ may overwrite objects $x_1, \ldots x_i$ and decide a different value violating the agreement property.

Let $\alpha_{i+1} = \alpha_i^v$, be the execution obtained after repeating $v = 2 \times (m-i) \times u_{i+1}$ times the previous scheduling, and $\Pi_{i+1} = \Pi_i^v$. By the pigeon holes theorem, there exists one of the $m - i$ objects in $X \setminus X_i$, that defines x_{i+1}, that is covered by at least $2 \times u_{i+1}$ processes of Π_{i+1}. 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. Moreover, $C(\alpha_{i+1})$ is $(\Pi_{i+1}, (u_i - v), X_i)$-covered, with $u_i - v = 2 \times l \times u_{i+1} \geq u_{i+1}$. Therefore, $C(\alpha_{i+1})$ is $(\Pi_{i+1}, u_{i+1}, X_{i+1})$-covered.

Finally, α_m is $(\Pi_m, \sim, p)$-separable, for some $\sim$ and p. In $C(\alpha_{i+1})$, at least $2 \leq u_m$ processes $p_i \neq p_j \in \Pi_m$ are about to write respectively v and w, to the last register, such that v was proposed by Process p_v and w was proposed by Process p_w, with $p_v \sim p_i \nsim p_j \sim p_w$. Then, p_i and p_j decide a different value, which violates agreement. □

Theorem 4. *Binary consensus has consensus number ∞_1^3.*

7 WINDOW REGISTERS HAVE 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 between n processes, for all n. A natural object to fill this degree of the hierarchy is the window registers, that can be seen as a composition of n-window registers for all n, each having consensus number n. This section proves that, indeed, window registers have consensus number ∞_1^2 (Theorem 5). We first need the preliminary result that WReg is not universal in MA_3 (Proposition 5).

It was already noted in [Afek et al. 2011] that having access to $\text{cons}^n\langle T \rangle$ objects for all n was not sufficient to solve consensus in MA_3. This section adapts the arguments to window registers. The proof relies on an extension of the classical valency notions to executions that only contain steps by processes with identifiers smaller than n (Definition 6).

Definition 6 (n-critical configuration). *Let α be an execution of a consensus algorithm. 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 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 8. *Any finite execution α such that $C(\alpha)$ is n-bivalent has an extension $\alpha\beta$ such that $C(\alpha\beta)$ is n-critical.*

Lemma 9. *In any n-critical configurations, $p_0, \ldots, p_{n-1}$ are about to write to the same k-WReg object, with $k \geq n$.*

Proposition 5. *MA_3[WReg] is not universal.*

Proof. Suppose there exists an algorithm A that solves consensus in MA_3[WReg]. 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 $n_0 \leq n_1 \leq n_2 \leq \ldots$ 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 $n_0 = 2$, and γ be the execution in which p_0 and p_1 propose 0 and 1 respectively. In a p_0-solo extension of γ, p_0 decides 0, and in a p_1-solo extension of γ, p_1 decides 1, so $C(\gamma)$ is n_0-bivalent. By Lemma 8, 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 9, in $C(\alpha_i)$, $p_0, \ldots, p_{n_i-1}$ are about to write to the same k-WReg object, with $k \geq n_i$. Let us pose $n_{i+1} = k + 1$. 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 8, 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 9, in $C(\alpha_{i+1})$, p_0 is about to write to a k'-WReg object, with $k' > k$. In particular, p_0 took its write step 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. This is a contradiction. □

Theorem 5. *Window registers have consensus number ∞_1^2.*

8 OBJECTS WITH CONSENSUS NUMBER ∞_2^3

As 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 Proposition 4, the proof of Proposition 6 builds 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 7 adapts the notion of coverage to take iterator stacks into account). The major difficulty is that iterator stacks cannot be overwritten by a finite number of processes, and the valency-based proof introduced in [Afek et al. 2011] cannot be adapted to a setting where binary consensus objects can be used in a critical configuration. Lemma 10 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 8, by the concept of blind extensions.

Definition 7 (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 8 (Blind extension). *Let α be a $(\Pi', \sim, p)$-partitioned execution. We say that $\alpha\beta$ is a blind extension of α if: 1) no process took steps in both α and β, and 2) 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 10. *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 $k \in \{0, \dots, m\}$, there exists a blind extension $\alpha\beta$ of α such that at least k different objects are write-covered in $C(\alpha\beta)$ by processes $r_1, \cdots, r_k$ that did not take steps in α.

Proof. We prove the claim 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 claim holds for some $k \in \{0, \dots, m-1\}$. We start the proof of $H(k+1)$ by proving a claim.

Claim 3. *There exists a blind extension $\alpha\beta$ of α such that at least k different objects are write-covered in $C(\alpha\beta)$ by processes $r_1, ..., r_k$ that did not take steps in α, and one more different object is covered in $C(\alpha\beta)$ by a process r_{k+1} that did not take steps in α.*

Proof. Suppose this claim is false. We build an infinite execution $\alpha\beta_0\beta_1\beta_2 \dots$ in which some process takes an infinite number of steps, such that each extension $\alpha\beta_0 \dots \beta_n$ is blind. Let w be the number of writes on iterator stacks in α, let $\alpha\beta_0 = \alpha\gamma_1 \dots \gamma_{w+2}$ 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. As we supposed Claim 3 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 $C(\alpha\beta_0)$. Let p be a process that did not take steps in $\alpha\beta_0$. Suppose we have built $\delta_n = \alpha\beta_0 \dots \beta_n$. We build β_{n+1} as follows.

- Suppose p 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 $\delta_{n+1} = \delta_n\zeta_1 \dots \zeta_{w'}\eta$ as follows: each ζ_l is the result of one invocation of the induction hypothesis $H(k)$. As we supposed Claim 3 was false, the set of write-covered objects in each ζ_l is Y. In particular, in $C(\delta_n\zeta_1 \dots \zeta_{w'})$, 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 p reads in y and gets one of the values written by one of these processes, which ensures the extension is blind.
- If p 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 p. 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 $\beta_0 \dots \beta_n$. As $i_\alpha \leq w$, p 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 $C(\delta_n)$, p 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 $\beta_0 \dots \beta_n$ or in α by some process $p' \sim p$. In all these cases, β_{n+1} is solely composed of the next step of p, which is a blind extension of δ_n.

Supposing Claim 3 is false, we built an execution in which process p takes an infinite number of steps, which contradicts wait-freedom and concludes the proof. □

Let us continue the proof of Lemma 10 by supposing that $H(k+1)$ is false. We build an infinite execution $\alpha\beta_0\beta_1\beta_2 \dots$ in which some process takes an infinite number of steps, and such that each extension $\alpha\beta_0 \dots \beta_n$ is blind.

Let w be the number of write operation 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 $\alpha\beta_0 = \alpha\gamma_1 \dots \gamma_{w'+1}$ such that each γ_l is the blind extension given by Claim 3. 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 to Y. Let Φ be the set of these $(w'+1)$ processes.

Let us suppose we have built a blind extension $\delta_n = \alpha\beta_0 \dots \beta_n$ 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 $p \in \Phi$ that did not read a value $v \neq \perp$ in an iterator stack $y \notin Y$ in δ_n. Such a process exists because 1) the hypothesis that $H(k+1)$ is false implies that no process wrote in an iterator stack $y \notin Y$ in $\beta_0 \dots \beta_n$, and 2) it is impossible to read a non-$\perp$ value in an iterator stack $y \notin Y$ more than $w' < |\Phi|$ times.

- Suppose p 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 $\delta_{n+1} = \delta_n\zeta_1 \dots \zeta_{2w''}\eta$ as follows. Let $\lambda_1 = \delta_n$ and $\lambda_i = \delta_n\zeta_1 \dots \zeta_{i-1}$ for $i > 1$. For each $i \in \{1, \dots, 2w''\}$, $\lambda_i\zeta_i$ 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 p. Such an extension exists by Claim 3 and the supposition that $H(k+1)$ is false. By the pigeon holes theorem, two cases are possible in $C(\delta_n\zeta_1 \dots \zeta_{2w''})$. If at least w'' new processes are about to read y, then η contains their read, and then the read by p returning $\perp$. Otherwise, at least w'' new processes are about to write in y, and η contains their write and the read by p, that returns a value written in η. In both cases, δ_{n+1} is blind.
- If p 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 p. As described in Claim 3, 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 p 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 $p' \sim p$. In all these cases, β_{n+1} is solely composed of the next step of p, which is a blind extension of δ_n.

Supposing $H(k+1)$ were false, We have built an execution in which some process 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.*

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

9 CONCLUSION

This paper explored 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. We extended the existing wait-free hierarchy by separating objects with an infinite consensus number into 5 categories, depending on their universality in the bounded, finite or infinite arrival models, and the need of an infinite memory allocation mechanism, or not. This paper raises several new open issues, that we detail thereafter.

In this paper, we supposed that processes shared an infinite memory. Although this assumption is central to the definition of the Turing Machine at the base of computer sciences, 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 [Zhang and Chen 2009]. An interesting open problem is the existence of a shared object with consensus number ∞_1^3 that does not have a polylogarithmic implementation of consensus in MA_2.

The objects we study in this paper, especially window registers and iterator stacks, are complex in the sense that they would typically require several memory locations to be implemented. Another open problem is the existence of special instructions that operate on a single (or fixed and small number of) memory locations, that fill each degree of the new hierarchy.

Finally, the example of an object of consensus number ∞_2^3 we exhibited in this paper is a composition of an object with consensus number ∞_2^2 and an object with consensus number ∞_1^3. It would be interesting to investigate if this is always the case, which can be split into two questions 1) Does there exist an object of consensus number ∞_2^3 that cannot be expressed as such a composition? 2) Conversely, does there exist two objects of consensus number ∞_2^2 and ∞_1^3 whose composition has consensus number ∞_3^3?

ACKNOWLEDGMENTS

This work was partially supported by the French ANR project 16-CE25-0005 O'Browser.

REFERENCES

Yehuda Afek, Adam Morrison, and Guy Wertheim. 2011. From bounded to unbounded concurrency objects and back. In *Proc. of the 30th Annual ACM Symposium on Principles of Distributed Computing, San Jose*. 119–128.

Marcos K Aguilera. 2004. A pleasant stroll through the land of infinitely many creatures. *ACM Sigact News* 35, 2 (2004), 36–59.

Hagit Attiya, Amotz Bar-Noy, Danny Dolev, David Peleg, and Rüdiger Reischuk. 1990. Renaming in an asynchronous environment. *J. ACM* 37, 3 (1990), 524–548.

Grégoire Bonin, Achour Mostéfaoui, and Matthieu Perrin. 2019. Wait-Free Universality of Consensus in the Infinite Arrival Model. In *International Symposium on Distributed Computing (DISC)*.

Keren Censor-Hillel, Erez Petrank, and Shahar Timnat. 2015. Help!. In *Proc. of the ACM Symposium on Principles of Distributed Computing*. 241–250.

Faith Ellen, Rati Gelashvili, Nir Shavit, and Leqi Zhu. 2016. A complexity-based hierarchy for multiprocessor synchronization. In *Proc. of the ACM Symposium on Principles of Distributed Computing*. 289–298.

Ivana Filipović, Peter O'Hearn, Noam Rinetzky, and Hongseok Yang. 2010. Abstraction for concurrent objects. *Theoretical Computer Science* 411, 51-52 (2010), 4379–4398.

Michael J. Fischer, Nancy A. Lynch, and Mike Paterson. 1985. Impossibility of Distributed Consensus with One Faulty Process. *J. ACM* 32, 2 (1985), 374–382.

Eli Gafni, Michael Merritt, and Gadi Taubenfeld. 2001. The concurrency hierarchy, and algorithms for unbounded concurrency. In *Proc. of the 20th ACM symposium on Principles of distributed computing*. 161–169.

Maurice Herlihy. 1991. Wait-free synchronization. *ACM TOPLAS* 13, 1 (1991), 124–149.

Maurice Herlihy, Sergio Rajsbaum, and Michel Raynal. 2013. Power and limits of distributed computing shared memory models. *Theor. Comput. Sci.* 509 (2013).

Maurice Herlihy and Nir Shavit. 2008. *The art of multiprocessor programming*. Morgan Kaufmann.

Maurice Herlihy and Jeannette M. Wing. 1990. Linearizability: A Correctness Condition for Concurrent Objects. *ACM TOPLAS* 12, 3 (1990), 463–492.

Michael C. Loui and Hosame H. Abu-Amara. 1987. Memory Requirements for Agreement Among Unreliable Asynchronous Processes. *Parallel and Distributed Computing* 4, 4 (1987), 163–183.

Achour Mostefaoui, Michel Raynal, and Frédéric Tronel. 2000. From binary consensus to multivalued consensus in asynchronous message-passing systems. *Inform. Process. Lett.* 73, 5-6 (2000), 207–212.

Matthieu Perrin, Achour Mostefaoui, and Claude Jard. 2016. Causal consistency: beyond memory. In *Proc. of the 21st ACM Symposium on Principles and Practice of Parallel Programming*. 1–12.

Serge A Plotkin. 1989. Sticky bits and universality of consensus. In *Proc. of the 8th ACM Symposium on Principles of distributed computing*. 159–175.

Michel Raynal. 2012. *Concurrent Programming - Algorithms, Principles, and Foundations*. Springer Science & Business Media.

Michel Raynal. 2017. Distributed Universal Constructions: a Guided Tour. *Bulletin of the EATCS* 121 (2017).

Gadi Taubenfeld. 2018. *Distributed Computing Pearls*. Morgan & Claypool Publishers. https://doi.org/10.2200/S00845ED1V01Y201804DCT014

Jialin Zhang and Wei Chen. 2009. Bounded cost algorithms for multivalued consensus using binary consensus instances. *Inform. Process. Lett.* 109, 17 (2009), 1005–1009.