Distributed Universality: Contention-Awareness, Wait-freedom, Object Progress, and Other Properties

Michel Raynal, Julien Stainer, Gadi Taubenfeld

Publications Internes de l'IRISA
ISSN : 2102-6327
PI 2014 – January 2014

Distributed Universality: Contention-Awareness, Wait-freedom, Object Progress, and Other Properties

Michel Raynal* ** Julien Stainer** Gadi Taubenfeld***

Abstract: A notion of a *universal construction* suited to distributed computing has been introduced by M. Herlihy in his celebrated paper "*Wait-free synchronization*" (ACM TOPLAS, 1991). A universal construction is an algorithm that can be used to wait-free implement any object defined by a sequential specification. Herlihy's paper shows that the basic system model, which supports only atomic read/write registers, has to be enriched with consensus objects to allow the design of universal constructions. The generalized notion of a k-*universal* construction has been recently introduced by Gafni and Guerraoui (CONCUR, 2011). A k-universal construction is an algorithm that can be used to simultaneously implement k objects (instead of just one object), with the guarantee that at least one of the k constructed objects progresses forever. While Herlihy's universal construction relies on atomic registers and consensus objects, a k-universal construction relies on atomic registers and k-simultaneous consensus objects (which are wait-free equivalent to k-set agreement objects in the read/write system model).

This paper significantly extends the universality results introduced by Herlihy and Gafni-Guerraoui. In particular, we present a k-universal construction which satisfies the following five desired properties, which are not satisfied by the previous k-universal construction: (1) among the k objects that are constructed, *at least* ℓ objects (and not just one) are guaranteed to progress forever; (2) the progress condition for processes is *wait-freedom*, which means that each correct process executes an infinite number of operations on each object that progresses forever; (3) if any of the k constructed objects stops progressing, all its copies (one at each process) stop in the same state; (4) the proposed construction is *contention-aware*, in the sense that it uses only read/write registers in the absence of contention; and (5) it is *generous* with respect to the *obstruction-freedom* progress condition, which means that each process is able to complete any one of its pending operations on the k objects if all the other processes hold still long enough. The proposed construction, which is based on new design principles, is called a (k, ℓ)-universal construction. It uses a natural extension of k-simultaneous consensus objects, called (k, ℓ)-simultaneous consensus objects ((k, ℓ)-SC). Together with atomic registers, (k, ℓ)-SC objects are shown to be necessary and sufficient for building a (k, ℓ)-universal construction, and, in that sense, (k, ℓ)-SC objects are (k, ℓ)-$universal$.

Key-words: Asynchronous read/write system, universal construction, consensus, k-set agreement, k-simultaneous consensus, wait-freedom, non-blocking, obstruction-freedom, contention-awareness, crash failures, state machine replication.

Universalité distribuée

Résumé : *Cet article explore la notion de construction universelle dans les systèmes distribués. Il présente une construction k-universelle wait-free qui s'adapte à la concurrence à partir d'objets k-consensus.*

Mots clés : *Système asynchrone read/write, construction universelle, consensus, k-accord ensembliste, k-consensus simultané, synchronisation sans attente, synchronisation non-bloquante, adaptivité à la concurrence, crash, réplication de machine d'état.*

---

* Institut Universitaire de France
** ASAP : équipe commune avec l'Université de Rennes 1 et Inria
*** The Interdisciplinary Center, Herzliya, Israel

1 Introduction

Asynchronous crash-prone read/write systems and the notion of a universal construction This paper considers systems made up of n sequential asynchronous processes that communicate by reading and writing atomic registers. Up to $n-1$ processes may crash unexpectedly. This is the basic $(n-1)$-resilient model, also called read/write *wait-free model*, and denoted here $\mathcal{ARW}_{n,n-1}[\emptyset]$. A fundamental problem encountered in this kind of systems consists in implementing any object, defined by a sequential specification, in such a way that the object behaves reliably despite process crashes.

Several progress conditions have been proposed for concurrent objects. The strongest, and most extensively studied condition, is wait-freedom. Wait-freedom guarantees that *every* process will always be able to complete its pending operations in a finite number of its own steps [17]. Thus, a *wait-free* implementation of an object guarantees that an invocation of an object operation may fail to terminate only when the invoking process crashes. The non-blocking progress condition (sometimes called lock-freedom) guarantees that *some* process will always be able to complete its pending operations in a finite number of its own steps [22]. Obstruction-freedom guarantees that a process will be able to complete its pending operations in a finite number of its own steps, if all the other processes "hold still" long enough [18]. Obstruction-freedom does not guarantee progress under contention.

It has been shown in [12, 17, 24] that the design of a general algorithm implementing *any* object defined by a sequential specification and satisfying the wait-freedom progress condition, is impossible in $\mathcal{ARW}_{n,n-1}[\emptyset]$. Thus, in order to be able to implement any such object, the model has to be enriched with basic objects whose computational power is stronger than atomic read/write registers [17].

Objects that can be used, together with registers, to implement any other object which satisfies a given progress condition *PC*, are called universal objects with respect to *PC*. Previous work provided algorithms, called *universal constructions*, based on universal objects, that transform sequential specifications of arbitrary objects into wait-free concurrent implementations of the same objects. It is shown in [17] that the *consensus* object is universal with respect to wait-freedom. A consensus object allows all the correct processes to reach a common decision based on their initial inputs. A consensus object is used in a universal construction to allow processes to agree –despite concurrency and failures– on a total order on the operations they invoke on the constructed object.

In addition to the universal construction of [17], several other wait-free universal constructions were proposed, which address additional properties. As an example, a universal construction is presented in [10], where "processes operating on different parts of an implemented object do not interfere with each other by accessing common base objects". Other additional properties have been addressed in [2, 11]. The notion of a universal construction has also been investigated in the context of transactional memories (e.g., [8, 9, 19, 20, 31] to cite a few).

From consensus to k-simultaneous consensus (or k-set agreement) in read/write systems k-Simultaneous consensus has been introduced in [1]. Each process proposes a value to k independent consensus instances, and decides on a pair (x, v) such that x is a consensus instance ($1 \leq x \leq k$), and v is a value proposed to that consensus instance. Hence, if the pairs (x, v) and (x, v') are decided by two processes, then $v = v'$.

k-Set agreement [7] is a simple generalization of consensus, namely, at most k different values can be decided on when using a k-set agreement object ($k = 1$ corresponds to consensus). It is shown in [1] that k-set agreement and k-simultaneous consensus have the same computational power in $\mathcal{ARW}_{n,n-1}[\emptyset]$. That is, each one can be solved in $\mathcal{ARW}_{n,n-1}[\emptyset]$ enriched with the other[1]. Hence, 1-simultaneous consensus is the same as consensus, while, for $k > 1$, k-simultaneous consensus is weaker than $(k-1)$-simultaneous consensus.

While the impossibility proof (e.g., [17, 24]) of building a wait-free consensus object in $\mathcal{ARW}_{n,n-1}[\emptyset]$ relies on the notion of valence introduced in [12], the impossibility to build a wait-free k-set agreement object (or equivalently a k-simultaneous consensus object) in $\mathcal{ARW}_{n,n-1}[\emptyset]$ relies on algebraic topology notions [5, 21, 30].

It is nevertheless possible to consider system models, stronger than the basic wait-free read/write model, enriched with consensus or k-simultaneous consensus objects. It follows that these enriched system models, denoted $\mathcal{ARW}_{n,n-1}[CONS]$ and $\mathcal{ARW}_{n,n-1}[k\text{-}SC]$ ($1 \leq k < n$), respectively, are computationally strictly stronger than the basic model $\mathcal{ARW}_{n,n-1}[\emptyset]$.

Universal construction for k objects An interesting question introduced in [14] by Gafni and Guerraoui is the following: what happens if, when considering the design of a universal construction, k-simultaneous consensus objects are considered instead of consensus objects? The authors claim that k-simultaneous consensus objects are *k-universal* in the sense that they allow to implement k deterministic concurrent objects, each defined by a sequential specification "with the guarantee that *at least one* machine remains highly available to all processes" [14]. In their paper, Gafni and Guerraoui focus on the replication of k state machines. They present

[1] This is no longer the case in asynchronous message-passing systems, namely k-simultaneous consensus is then strictly stronger than k-set agreement (as shown using different techniques in [6, 29]).

a k-universal construction, based on the replication –at every process– of each of the k state machines. This construction is presented in appendix A.

Contributions This paper is focused on *distributed universality*, namely it presents a very general universal construction for a set of n processes that access k concurrent objects, each defined by a sequential specification on total operations. An operation on an object is "total" if, when executed alone, it always returns [22]. This construction is based on a generalization of the k-simultaneous consensus object (see below). The noteworthy features of this construction are the following.

- At least ℓ among the k objects progress forever, $1 \leq \ell \leq k$. This means that an infinite number of operations is applied to each of these ℓ objects. This set of ℓ objects is not predetermined, and depends on the execution.
- The progress condition associated with the processes is wait-freedom. That is, a process that does not crash executes an infinite number of operations on each object that progresses forever.
- An object stops progressing when no more operations are applied to it. The construction guarantees that, when an object stops progressing, all its copies (one at each process) stop in the same state.
- The construction is *contention-aware*. This means that the overhead introduced by using synchronization objects other than atomic read/write registers is eliminated when there is no contention during the execution of an operation (i.e., interval contention). In the absence of contention, a process completes its operations by accessing only read/write registers[2]. Algorithms which satisfy the contention-awareness property have been previously presented in [3, 26, 27, 32].
- The construction is *generous* with respect to *obstruction-freedom*. This means that each process is able to complete its pending operations on all the k objects each time all the other processes hold still long enough. That is, if once and again all the processes except one hold still long enough, then all the k objects, and not just ℓ objects, are guaranteed to always progress.[3]

This new universal construction is consequently called a *contention-aware* obstruction-free-generous wait-free (k, ℓ)-universal construction. Differently, the universal construction presented in [14] is a $(k, 1)$-universal construction and is neither contention-aware, nor generous with respect to obstruction-freedom. Moreover, this construction suffers from the following limitations: (a) it does not satisfy wait-freedom progress, but only non-blocking progress (i.e., infinite progress is guaranteed for only one process); (b) in some scenarios, an operation that has been invoked by a process can (incorrectly) be applied twice, instead of just once; and (c) the last state of the copies (one per process) of an object on which no more operations are being executed can be different at distinct processes. While issue (b) can be fixed (see Appendix A), we do not see how to modify the construction from [14] to overcome drawback (c).

When considering the special case $k = \ell = 1$, Herlihy's construction is wait-free $(1, 1)$-universal [17], but differently from ours, it does not satisfy the contention-awareness property.

To ensure the progress of at least ℓ of the k implemented objects, the proposed construction uses a new synchronization object, that we call (k, ℓ)-simultaneous consensus object, which is a simple generalization of the k-simultaneous consensus object. This object type is such that its $(k, 1)$ instance is equivalent to k-simultaneous consensus, while its (k, k) instance is equivalent to consensus. Thus, when added to the basic $\mathcal{ARW}_{n,n-1}[\emptyset]$ system model, (k, ℓ)-simultaneous consensus objects add computational power. The paper shows that (k, ℓ)-simultaneous consensus objects are both *necessary and sufficient* to ensure that at least ℓ among the k objects progress forever.

From a software engineering point of view, the proposed (k, ℓ)-universal construction is built in a modular way. First a non-blocking $(k, 1)$-universal construction is designed, using k-simultaneous consensus objects and atomic registers. Interestingly, its design principles are different from the other universal constructions we are aware of. Then, this basic construction is extended to obtain a contention-aware $(k, 1)$-universal construction, and then a wait-free contention-aware $(k, 1)$-universal construction. Finally, assuming that the system is enriched with (k, ℓ)-simultaneous consensus objects, $1 \leq \ell \leq k$, instead of k-simultaneous consensus objects, we obtain a contention-aware wait-free (k, ℓ)-universal construction. During the modular construction, we make sure that the universal construction implemented at each stage is also generous with respect to obstruction-freedom.

[2]Let us recall that, in *worst case* scenarios, hardware operations such as compare&swap() can be $1000\times$ more expensive that read or write.

[3]*Generosity* is a general notion. Intuitively, an algorithm is *generous* with respect to a given condition C, if, whenever C is satisfied, the algorithm does more than what it is required to do in normal circumstances. The condition C specifies the "exceptional" circumstances under which the algorithm does "more". These "exceptional" circumstances depend on the underlying system behavior. They can be a specific progress condition (as done in this paper), or the occurrences of specific synchrony/asynchrony/failures patterns. The notions of a *generous algorithm* and of an *indulgent algorithm* (investigated in [16]) can be seen as "dual" one from the other, in the following sense. Indulgence allows an algorithm to do less (more precisely not to terminate) when the underlying system (captured as an underlying failure detector) is misbehaving (i.e., doing less), while generosity forces an algorithm to do more when the underlying system is doing more than what is normally expected.

Roadmap The paper is made up of 5 sections. Section 2 presents the computation models and the specific objects used in the paper. Section 3 presents a non-blocking $(k, 1)$-universal construction. Then Section 4 extends it so that it satisfies contention-awareness, wait-freedom, and the progress of at least ℓ out of the k constructed objects. This section shows also that (k, ℓ)-simultaneous consensus objects are necessary and sufficient for the design of (k, ℓ)-universal constructions. Due to page limitation, the presentation of an interesting simple variant of the general universal construction which is an obstruction-free $(1, 1)$-universal construction based on atomic registers only, is presented in Appendix F. Definitions and notions which can be used to establish a (k, ℓ)-universality theory are presented in Appendix G. Moreover, all proofs are in appendices. Section 5 concludes the paper.

2 Basic and Enriched Models, and Wait-free Linearizable Implementation

2.1 Basic read/write model and enriched model

The basic model is the wait-free asynchronous read/write model denoted $\mathcal{ARW}_{n,n-1}[\emptyset]$ presented in the introduction (see also [4, 25, 28]). The processes are denoted p_1, ..., p_n. Considering a run, a process is *faulty* if it crashes during the run, otherwise it is *correct*.

In addition to atomic read/write registers [23], two other types of objects are used. The first type does not add computational power, but provides processes with a higher abstraction level. The other type adds computational power to the basic system model $\mathcal{ARW}_{n,n-1}[\emptyset]$.

Adopt-commit object The adopt-commit object has been introduced in [13]. An adopt-commit object is a one-shot object that provides the processes with a single operation denoted propose(). This operation takes a value as an input parameter, and returns a pair (tag, v). The behavior of an adopt-commit object is formally defined as follows:

- Validity.
  - Result domain. Any returned pair (tag, v) is such that (a) v has been proposed by a process and (b) $tag \in \{commit,\ adopt\}$.
  - No-conflicting values. If a process p_i invokes propose(v) and returns before any other process p_j has invoked propose(v') with $v' \neq v$, then only the pair $(commit, v)$ can be returned.
- Agreement. If a process returns $(commit, v)$, only the pairs $(commit, v)$ or $(adopt, v)$ can be returned.
- Termination. An invocation of propose() by a correct process always terminates.

Let us notice that it follows from the "no-conflicting values" property that, if a single value v is proposed, then only the pair $(commit, v)$ can be returned. Adopt-commit objects can be wait-free implemented in $\mathcal{ARW}_{n,n-1}[\emptyset]$ (e.g., [13, 28]). Hence, they provide processes with a higher abstraction level than read/write registers.

k-Simultaneous consensus object A k-simultaneous consensus (k-SC) object is a one-shot object that provides the processes with a single operation denoted propose(). This operation takes as input parameter a vector of size k, each entry containing a value, and returns a pair (x, v). The behavior of a k-simultaneous consensus object is formally defined as follows:

- Validity. Any pair (x, v) that is returned by a process p_i is such that (a) $1 \leq x \leq k$ and (b) v has been proposed by a process in the x-th entry of its input vector before p_i decides.
- Agreement. If a process returns (x, v) and another process returns (y, v'), and $x = y$, then $v = v'$.
- Termination. An invocation of propose() by a correct process always terminates.

Let $\mathcal{ARW}_{n,n-1}[k\text{-}SC]$ denote $\mathcal{ARW}_{n,n-1}[\emptyset]$ enriched with k-SC objects. It is shown in [1] that a k-SC object and a k-set agreement (k-SA) object are wait-free equivalent in $\mathcal{ARW}_{n,n-1}[\emptyset]$. This means that a k-SC object can be built in $\mathcal{ARW}_{n,n-1}[k\text{-}SA]$, and a k-SA object can be built in $\mathcal{ARW}_{n,n-1}[k\text{-}SC]$.

2.2 Correct object implementation

Let us consider n processes that access k concurrent objects, each defined by a deterministic sequential specification. The sequence of operations that p_i wants to apply to an object m, $1 \leq m \leq k$, is stored in the local infinite list $my\_list_i[m]$, which can be defined statically or dynamically (in that case, the next operation issued by a process p_i on an object m, can be determined from p_i's view of the global state). It is assumed that the processes are well-formed: no process invokes a new operation on an object m before its previous operation on m has terminated.

Wait-free linearizable implementation An implementation of an object m by n processes is wait-free linearizable if it satisfies the following properties.

- Validity. If an operation op is executed on object m, then $op \in \cup_{1 \leq i \leq n} my\_list_i[m]$, and all the operations of $my\_list_i[m]$ which precede op have been applied to object m.
- No-duplication. Any operation op on object m invoked by a process is applied at most once to m. We assume that all the invoked operations are unique.
- Consistency. Any n-process execution produced by the implementation is linearizable [22].
- Termination (wait-freedom). If a process does not crash, it executes an infinite number of operations on at least one object.

Weaker progress conditions In some cases, the following two weaker progress conditions are considered.

- The *non-blocking* progress condition [22] guarantees that there is at least one process that executes an infinite number of operations on at least one object.
- The *obstruction-freedom* progress condition [18] guarantees that any correct process can complete its operations if it executes in isolation for a long enough period (i.e., there is a long enough period during which the other processes stop progressing).

3 Part 1: A New Non-blocking k-Universal Construction

As mentioned in the Introduction, the construction is done incrementally. In this section, we present and prove the correctness of a non-blocking k-universal construction, based on new design principles (as far as we know). This construction is built in the enriched model $\mathcal{ARW}_{n,n-1}[k\text{-}SC]$. In Section 4, we extend the construction, without requiring additional computational power, to obtain the contention-awareness property, and the wait-freedom progress condition (i.e., *each* correct process can always execute and completes its operations on any object that progresses forever). Then (k, ℓ)-SC objects are introduced (which are a natural generalization of k-SC objects), and are used to design a (k, ℓ)-universal construction which ensures that least ℓ objects progress forever. In Section 4, we also show that (k, ℓ)-SC objects are necessary and sufficient to obtain a (k, ℓ)-universal construction.

3.1 A new non-blocking k-universal construction: data structures

The following objects are used by the construction. Identifiers with upper case letters are used for shared objects, while identifiers with lower case letters are used for local variables.

Shared objects

- $kSC[1..]$: infinite list of of k-simultaneous consensus objects; $kSC[r]$ is the object used at round r.
- $AC[1..][1..k]$: infinite list of vectors of k adopt-commit objects; $AC[r][m]$ is the adopt-commit object associated with the object m at round r.
- $GSTATE[1..n]$ is an array of SRMW (single-writer/multi-readers) atomic registers; $GSTATE[i]$ can be written only by p_i. Moreover, the register $GSTATE[i]$ is made up of an array with one entry per object, such that $GSTATE[i][m]$ is the sequence of operations that have been applied to the object m, as currently know by p_i; it is initialized to ϵ (the empty sequence).

Local variables at process p_i

- r_i: local round number (initialized to 0).
- $g\_state_i[1..n]$: array used to save the values read (non-atomically) from $GSTATE[1..n]$.
- $oper_i[1..k]$: vector such that $oper_i[m]$ contains the operation that p_i is proposing to a k-SC object for the object m (as we will see in the algorithm, this operation was not necessarily issued by p_i).
- $my\_op_i[1..k]$: vector of operations such that $my\_op_i[m]$ is the last operation that p_i wants to apply to the object m (hence $my\_op_i[m] \in my\_list_i[m]$).
- $\ell\_hist_i[1..k]$: vector with one entry per object, such that $\ell\_hist_i[m]$ is the sequence of operations defining the history of object m, as known by p_i. Each $\ell\_hist_i[m]$ is initialized to ϵ. The function append() is used to add an element at the end of a sequence $\ell\_hist_i[m]$.
- $tag_i[1..k]$ and $ac\_op_i[1..k]$: arrays such that, for each object m, $tag_i[m]$ and $ac\_op_i[m]$ are used to save the pairs $(tag, operation)$ returned by the last invocation of $AC[r][m]$, during round r.
- $output_i[1..k]$: vector such that $output_i[m]$ contains the result of the last operation invoked by p_i on the object m (this is the operation saved in $my\_op_i[m]$).

Without loss of generality, it is assumed that each object operation returns a result, which can be "ok" when there is no object-dependent result to be returned (as with the stack operation push() or the queue operation enqueue()).

3.2 Eliminating full object histories

Each entry m of the previous atomic variables $GSTATE[i]$, i.e., $GSTATE[i][m]$, and each local variable $\ell\_hist_i[m]$, contain sequences of operations successfully applied to m, as known by p_i. This implementation has been chosen for its simplicity. A more space efficient implementation of these objects is described in Appendix C (each object is represented by its last state, and each process has to manage additional sequence numbers).

3.3 A new non-blocking $(k, 1)$-universal construction: algorithm

To simplify the presentation, it is assumed that each operation invocation is unique. This can be easily realized by associating an identity (process id, sequence number) with each operation invocation. In the following, the term "operation" is used as an abbreviation for "operation execution".

The function $\mathsf{next}()$ is used by a process p_i to access the sequence of operations $my\_list_i[m]$. The x-th invocation of $my\_list_i[m].\mathsf{next}()$ returns the x-th element of this list.

for each $m \in \{1, \ldots, k\}$ **do** $my\_op_i[m] \leftarrow my\_list_i[m].\mathsf{next}()$; $oper_i[m] \leftarrow my\_op_i[m]$ **end for**.

repeat forever
(1) $r_i \leftarrow r_i + 1$;
(2) $(ksc\_obj, ksc\_op) \leftarrow kSC[r_i].\mathsf{propose}(oper_i[1..k])$;
(3) $(tag_i[ksc\_obj], ac\_op_i[ksc\_obj]) \leftarrow AC[r_i][ksc\_obj].\mathsf{propose}(ksc\_op)$;
(4) **for each** $m \in \{1, \ldots, k\} \setminus \{ksc\_obj\}$ **do** $(tag_i[m], ac\_op_i[m]) \leftarrow AC[r_i][m].\mathsf{propose}(oper_i[m])$ **end for**;

(5) **for each** $j \in \{1, \ldots, n\}$ **do** $g\_state_i[j] \leftarrow GSTATE[j]$ **end for**; % the read of each $GSTATE[j]$ is atomic %
(6) **for each** $m \in \{1, \ldots, k\}$ **do**
(7) $\ell\_hist_i[m] \leftarrow$ longest history of $g\_state_i[1..n][m]$ containing $\ell\_hist_i[m]$;
(8) **if** $(my\_op_i[m] \in \ell\_hist_i[m])$ % my operation was completed %
(9) **then** $output_i[m] \leftarrow \mathsf{compute\_output}(my\_op_i[m], \ell\_hist_i[m])$;
(10) return $\{(m, my\_op_i[m], output_i[m])\}$ to the upper layer;
(11) $my\_op_i[m] \leftarrow my\_list[m].\mathsf{next}()$
(12) **end if**
(13) **end for**;

(14) $res \leftarrow \emptyset$;
(15) **for each** $m \in \{1, \ldots, k\}$ **do**
(16) **if** $(ac\_op_i[m] \notin \ell\_hist_i[m])$ % operation was not completed %
(17) **then if** $(tag_i[m] = commit)$ % complete the operation %
(18) **then** $\ell\_hist_i[m] \leftarrow \ell\_hist_i[m].\mathsf{append}(ac\_op_i[m])$;
(19) **if** $(ac\_op_i[m] = my\_op_i[m])$ % my operation was completed %
(20) **then** $output_i[m] \leftarrow \mathsf{compute\_output}(ac\_op_i[m], \ell\_hist_i[m])$;
(21) $res \leftarrow res \cup \{(m, my\_op_i[m], output_i[m])\}$;
(22) $my\_op_i[m] \leftarrow my\_list[m].\mathsf{next}()$
(23) **end if**;
(24) $oper_i[m] \leftarrow my\_op_i[m]$
(25) **else** $oper_i[m] \leftarrow ac\_op_i[m]$ % $tag_i[m] = adopt$ %
(26) **end if**
(27) **else** $oper_i[m] \leftarrow my\_op_i[m]$ % $ac\_op_i[m] \in \ell\_hist_i[m]$ %
(28) **end if**
(29) **end for**;

(30) $GSTATE[i] \leftarrow \ell\_hist_i[1..k]$; % globally update my current view %
(31) **if** $(res \neq \emptyset)$ **then** return res to the upper layer **end if**
end repeat.

Figure 1: Basic Non-Blocking Generalized $(k, 1)$-Universal Construction (code for p_i)

Initialization The algorithm implementing the k-universal construction is presented in Figure 1. For each object $m \in \{1, ..., k\}$, a process p_i initializes both the variables $my\_op_i[m]$ and $oper_i[m]$ to the first operation that it wants to apply to m. Process p_i then enters an infinite loop.

Repeat loop: using the round r objects $kSC[r]$ and $AC[r]$ (lines 1-4) After it has increased its round number, a process p_i invokes the k-simultaneous consensus object $kSC[r]$ to which it proposes the operation vector $oper_i[1..n]$, and from which it obtains the pair denoted (ksc_obj, ksc_op); ksc_op is an operation proposed by some process for the object ksc_obj (line 2). Process p_i then invokes the adopt-commit object $AC[r][ksc\_obj]$ to which it proposes the operation output by $kSC[r]$ for the object ksc_op (line 3). Finally, for all the other objects $m \neq ksc\_obj$, p_i invokes the adopt-commit object $AC[r][m]$ to which it proposes $oper_i[m]$ (line 4). As already indicated, the tags and the commands defined by the vector of pairs output by the adopt-commit objects $AC[r]$ are saved in the vectors $tag_i[1..k]$ and $ac\_op_i[1..k]$, respectively. (While expressed differently, these four lines are the only part which is common to this construction and the one presented in [14].)

The aim of these lines is to implement a filtering mechanism such that (a) for each object, at most one operation can be committed at some processes, and (b) there is at least one object for which an operation is committed at some processes.

Repeat loop: returning local results (lines 5-13) After having used the additional power supplied by $kSC[r]$, a process p_i first obtains asynchronously the value of $GSTATE[1..n]$ (line 5) to learn an "as recent as possible" consistent global state, which is saved in $g\_state_i[1..n]$. Then, for each object m (lines 6-13), p_i computes the maximal local history of the object m which contains $\ell\_hist_i[m]$ (line 7). (Let us notice that $g\_state_i[i][m]$ is $\ell\_hist_i[m]$.) This corresponds to the longest history in the n histories $g\_state_i[1][m]$, ..., $g\_state_i[n][m]$ which contains $\ell\_hist_i[m]$. If there are several longest histories, they all are equal as we will see in the proof. If the last operation it has issued on m, namely $my\_op_i[m]$, belongs to this history (line 8), some process has executed this operation on its local copy of m. Process p_i computes then the corresponding output (line 9), locally returns the triple $(m, my\_op_i[m], output_i[m])$ (line 10), and defines its next local operation to apply to the object m (line 11).

The function compute_output(op, h) (used at lines 9 and 20) computes the result returned by op applied to the state of the corresponding object m (this state is captured by the prefix of the history h of m ending just before the operation op).

Repeat loop: trying to progress on machines (lines 14-29) Then, for each object m, $1 \leq m \leq k$, p_i considers the operation $ac\_op_i[m]$. If this operation belongs to its local history $\ell\_hist_i[m]$ (the predicate of line 16 is then false), it has already been locally applied; p_i consequently assigns $my\_op_i[m]$ to $oper_i[m]$, where is saved its next operation on the object m (line 27).

If $ac\_op_i[m] \notin \ell\_hist_i[m]$ (line 16), the behavior of p_i depends on the fact that the tag of $ac\_op_i[m]$ is $commit$ or $adopt$. If the tag is $adopt$ (the predicate of line 17 is then false), p_i defines $ac\_op_i[m]$ as the next operation it will propose for the object m, which is saved in $oper_i[m]$ (line 25): it "adopts" $ac\_op_i[m]$. If the tag is $commit$ (line 17), p_i adds (applies) the operation $ac\_op_i[m]$ to its local history (line 18). Moreover, if $ac\_op_i[m]$ has been issued by p_i itself (i.e., $ac\_op_i[m] = my\_op_i[m]$, line 19), p_i computes the result locally returned by $ac\_op_i[m]$ (line 20), adds this result to the set of results res (line 21), defines its next local operation to apply to the object m (line 22). Finally, p_i assigns $my\_op_i[m]$ to $oper_i[m]$ (line 24).

Repeat loop: making public its progress (lines 30-31) Finally, p_i makes public its current local histories (one per object) by writing them in $GSTATE[i]$ (line 30), and returns local results if any (line 31). It then progresses to the next round.

3.4 A new non-blocking k-universal construction: proof

Lemma 1 $\forall\, i, m$: $(op \in GSTATE[i][m]) \Rightarrow (\exists\, j : op \in my\_list_j[m])$ *(i.e., if an operation op is applied to an object m, then op has been proposed by a process).*

Lemma 2 $\forall\, i, j, m : (op \in my\_list_j[m]) \Rightarrow$ *(op appears at most once in* $GSTATE[i][m]\big)$ *(i.e., an operation is executed at most once).*

The sequence $(op_r^m)_{r\geq 1}$ of committed operations According to the specification of the adopt-commit object, for any round r and any object m there is at most one operation returned with the tag $commit$ by the object $AC[r][m]$ to some processes. Let op_r^m denote this unique operation if at least one process obtains a pair with the tag $commit$, and let op_r^m be $\perp$ if all the pairs returned by $AC[r][m]$ contain the tag $adopt$.

From the sequence $(op_r^m)_{r\geq 1}$ to the notion of valid histories Considering an execution of the algorithm of Figure 1, the following lemmas show that, for any process p_i and any object m, all the sequences of operations appearing in $\ell\_hist_i[m]$ are finite prefixes of a unique valid sequence depending only on the sequence $(op_r^m)_{r\geq 1}$ of committed operations.

More precisely, given a sequence $(op_r^m)_{r\geq 1}$, a history $(vh_x^m)_{1\leq x\leq xmax}$ is *valid* if it is equal to a sequence $(op_r^m)_{1\leq r\leq R}$ from which the $\perp$ values and the repetitions have been removed. More formally, $(vh_x^m)_{1\leq x\leq xmax}$ is valid if there is a round number R and a strictly increasing function $\sigma : \{1, \ldots, xmax\} \rightarrow \{1, \ldots, R\}$ such that for all x in $\{1, \ldots, xmax\}$: (a) $vh_x^m = op_{\sigma(x)}^m$, (b) $vh_x^m \neq \perp$, (c) for all x in $\{1, \ldots, xmax-1\}$: $vh_x^m \neq vh_{x+1}^m$, and (d) the sets $\{vh_1^m, ..., vh_{xmax}^m\}$ and $\{op_1^m, ..., op_R^m\} \setminus \{\perp\}$ are equal.

Let us remark that this definition has two consequences: (i) the value of R for which item (d) is verified defines unambiguously the sequence $(vh_x^m)_{1\leq x\leq xmax}$ (and accordingly this sequence is denoted $VH^m(R)$ in the following), and (ii) for any two valid histories $(vh_x^m)_{1\leq x\leq xmax1}$ and $(vh_x^m)_{1\leq x\leq xmax2}$, one is a prefix of the other.

Lemma 3 *For any process p_i and any object m, at any time the local history $\ell\_hist_i[m]$ is valid.*

The execution on an object m of an operation op, issued by a process p_i, starts when the process p_i proposes op to a k-simultaneous consensus object $kSC[-][m]$ for the first time (i.e., when p_i makes op public), and terminates when a set res including $(m, op, output[m])$ is returned by p_i at line 10 or line 31. The next lemma shows that any execution is linearizable.

Lemma 4 *The execution of an operation op issued by a process p_i on an object m can be linearized at the first time at which a process p_j writes into $GSTATE[j][m]$ a local history $\ell\_hist_j[m]$ such that $op \in \ell\_hist_j[m]$.*

Lemma 5 $\forall\ r \geq 1$, *there is a process p_i such that at least one operation op output by $kSC[r]$.propose() at p_i (line 2) is such that the invocation of $AC[r][-]$.propose() by p_i returns $(commit, op)$ (line 3 or 4).*

Lemma 6 *There is at least one object on which an infinite number of operations are executed.*

It follows from the previous lemma, and the fact that there is a bounded number of processes, that at least one process executes an infinite number of its operations on an object. Hence the following corollary.

Corollary 1 *The algorithm is non-blocking.*

Theorem 1 *The algorithm of Figure 1 is a non-blocking linearizable $(k, 1)$-universal construction.*

Generosity wrt obstruction-freedom We observe that the construction of Figure 1 is also obstruction-free (k, k)-universal. That is, the construction guarantees that each process will be able to complete all its pending operations in a finite number of steps, if all the other processes "hold still" long enough. Thus, if once in a while all the processes except one "hold still" long enough, then all the k objects (and not "at least one") are guaranteed to always make progress.

4 Part 2: A Contention-Aware Wait-free (k, ℓ)-Universal Construction

4.1 A Contention-aware non-blocking k-universal construction

Contention-aware universal construction A *contention-aware* universal construction (or object) is a construction (object) in which the overhead introduced by synchronization primitives which are different from atomic read/write registers (like k-SC objects) is eliminated in executions when there is no contention. When a process invokes an operation on a contention-aware universal construction (object), it must be able to complete its operation by accessing only read/write registers in the absence of contention. Using other synchronization primitives is permitted only when there is contention. (This notion is close but different from the notion of *contention-sensitiveness* introduced in [32].)

A contention-aware non-blocking $(k, 1)$-universal construction A contention-aware $(k, 1)$-universal construction is presented in Figure 2. At each round r, it uses two adopt-commit objects per constructed object m, namely $AC[2r_i - 1][m]$ and $AC[2r_i][m]$, instead of a single one. When considering the basic construction of Figure 1, the new lines are prefixed by N, while modified lines are postfixed by M.

A process p_i first invokes, for each object m, the adopt-commit object $AC[2r_i - 1][m]$ to which it proposes $oper_i[m]$ (new line N1). Its behavior depends then on the number of objects for which it has received the tag $commit$. If it has obtained the tag $commit$ for all the objects m (the test of the new line N2 is then false), p_i proceeds directly to the code defined by the lines 5- 31 of the basic construction described in Figure 1, thereby skipping the invocation of the synchronization object $kSC[r]$ associated with round r.

Otherwise, the test of the new line N2 is true and there is at least one object for which p_i has received the tag $adopt$. This means that there is contention. In this case, the behavior of p_i is similar to the lines 2-4 of the basic algorithm where, at lines 2 and 4, the input parameter $oper_i[m]$ is replaced by the value of $ac\_op_i[m]$ obtained at line N1 (the corresponding lines are denoted 2M and 4M). Moreover, at line 3, r_i is replaced by $2r_i$ (new line 3M). It is possible to reduce the number of uses of underlying k-SC synchronization objects. Such an improvement is described in Appendix E.

Interestingly, for the case of $k = 1$, the above universal construction is the first known *contention-aware* $(1, 1)$-universal construction.

Theorem 2 *The algorithm of Figure 2 is a non-blocking contention-aware $(k, 1)$-universal construction.*

for each $m \in \{1, \ldots, k\}$ **do** $my\_op_i[m] \leftarrow my\_list_i[m]$.next(); $oper_i[m] \leftarrow my\_op_i[m]$ **end for**.

repeat forever
(1) $r_i \leftarrow r_i + 1$;
(N1) **for each** $m \in \{1, \ldots, k\}$ **do** $(tag_i[m], ac\_op_i[m]) \leftarrow AC[2r_i - 1][m]$.propose($oper_i[m]$) **end for**;
(N2) **if** $(\exists\, m \in \{1, \ldots, k\} :\ tag_i[m] = adopt)$ **then**
(2M) $(ksc\_obj, ksc\_op) \leftarrow kSC[r_i]$.propose($ac\_op_i[1..k]$);
(3M) $(tag_i[ksc\_obj], ac\_op_i[ksc\_obj]) \leftarrow AC[2r_i][ksc\_obj]$.propose($ksc\_op$);
(4M) **for each** $m \in \{1, \ldots, k\} \setminus \{ksc\_obj\}$ **do** $(tag_i[m], ac\_op_i[m]) \leftarrow AC[2r_i][m]$.propose($ac\_op_i[m]$) **end for**
(N3) **end if**;
lines 5- 31 of the construction of Figure 1
end repeat.

Figure 2: Contention-aware Non-Blocking $(k, 1)$-Universal Construction (code for p_i)

4.2 On the process side: from non-blocking to wait-freedom

The aim here is to ensure that each correct process executes an infinite number of operations on each object that progresses forever. As far as the progress of objects is concerned, it is important to notice that, while Lemma 6 shows that there is always at least one object that progresses forever, it is possible that, in a given execution, several objects progress forever.

Going from non-blocking to wait-freedom requires to add a helping mechanism to the basic non-blocking construction. To that end, the following array of atomic registers is introduced.

- $LAST\_OP[1..n, 1..m]$: matrix of atomic SWMR (single-writer/multi-readers) registers such that $LAST\_OP[i, m]$ contains the last operation of my_list_i invoked by p_i. Initialized to $\perp$, such a register is updated each time p_i invokes my_list_i.next() (initialization, line 11n and line 22). So, we assume that $LAST\_OP[i, m]$ is implicitly updated by p_i when it invokes the function next().

Then, for each object m, the lines 24 and 27 where is defined $oper_i[m]$ (namely, the proposal for the constructed object m submitted by p_i to the next k-SC object) are replaced by the following lines ($|s|$ denotes the size of the sequence s).

(L1) $j \leftarrow |\ell\_hist_i[m]| \bmod n + 1$; $next\_prop\_m \leftarrow LAST\_OP[j, m]$;
(L2) **if** $next\_prop\_m \notin (\{\perp\} \cup \ell\_hist_i[m])$
(L3) **then** $oper_i[m] \leftarrow next\_prop\_m$
(L4) **else** $oper_i[m] \leftarrow my\_op_i[m]$
(L5) **end if**.

This helping mechanism is close to the one proposed in [17]. It uses, for each object m, a simple round-robin technique on the process identities, computed from the current state of m as known by p_i, i.e., from $\ell\_hist_i[m]$. More precisely, the helping mechanism uses the number of operations applied so far to m (to p_i's knowledge) in order to help the process p_j such that $j = |\ell\_hist_i[m]| \bmod n + 1$ (line L1). To that end, p_i proposes the last operation issued by p_j on m (line L3) if (a) there is such an operation, and (b) this operation has not yet been appended to its local history of m (predicate of line L2). This operation has been registered in $LAST\_OP[j, m]$ when p_j executed its last invocation of $my\_list_j[m]$.next(). If the predicate of line L2 is not satisfied, p_i proceed as in the basic algorithm (line L4).

Theorem 3 *When replacing the lines* 24 *and* 27 *by lines* L1-L5*, the algorithms of Figure* 1 *and Figure 2 are wait-free linearizable* $(k, 1)$*-universal constructions.*

Let us remark that requiring wait-freedom only for a subset of correct processes, or only for a subset of objects that progress forever is not interesting, as wait-freedom for both (a) all correct processes, and (b) all the objects that progress forever, does not require additional computing power.

4.3 On the object side: from one to ℓ objects that always progress

Definition: (k, ℓ)**-Simultaneous consensus** Let (k, ℓ)-simultaneous consensus (in short (k, ℓ)-SC), $1 \leq \ell \leq k$, be a strengthened form of k-simultaneous consensus where (instead of a single pair) a process decides on ℓ pairs $(x_1, v_1), \ldots, (x_\ell, v_\ell)$ (all different in their first component). The agreement property is the same as for a k-SC object, namely, if (x, v) and (x, v') are pairs decided by two processes, then $v = v'$.

Notations Let (k,ℓ)-UC be any algorithm implementing the k-universal construction where at least ℓ objects always progress[4]. Let $\mathcal{ARW}_{n,n-1}[(k,\ell)\text{-}SC]$ be $\mathcal{ARW}_{n,n-1}[\emptyset]$ enriched with (k,ℓ)-SC objects, and $\mathcal{ARW}_{n,n-1}[(k,\ell)\text{-}UC]$ be $\mathcal{ARW}_{n,n-1}[\emptyset]$ enriched with a (k,ℓ)-UC algorithm.

A contention-aware wait-free (k,ℓ)universal construction A contention-aware wait-free (k,ℓ)-UC algorithm can be implemented as follows on top of $\mathcal{ARW}_{n,n-1}[(k,\ell)\text{-}SC]$. This algorithm is the algorithm of Figure 2, where lines 24 and 27 are replaced by the lines L1-L5 introduced in Section 4.2, and where the lines 2M, 3M, and 4M, are modified as follows (no other line is added, suppressed, or modified).

- Line 2M: the k-simultaneous consensus objects are replaced by (k,ℓ)-simultaneous consensus objects, Hence, the result returned to a process is now a set of ℓ pairs, all different in their first component, denoted $\{(ksc\_obj_1, ksc\_op_1), ..., (ksc\_obj_\ell, ksc\_op_\ell)\}$. Let L be the corresponding set of ℓ different objects, i.e., $L = \{ksc\_obj_1, ..., ksc\_obj_\ell\}$. As already indicated, two different processes can be returned different sets of ℓ pairs.
- Line 3M: process p_i executes this line for each object $m \in L$. These ℓ invocations of the adopt-commit object (i.e., $AC[2r_i][ksc\_obj_x]$.propose(ksc_op_x), $1 \le x \le \ell$) can be executed in parallel, which means in any order. Let us notice that if several processes invokes $AC[2r_i][ksc\_obj_x]$.propose() on the same object ksc_obj_x, they invoke it with the same operation ksc_op_x.
- Line 4M: $AC[2r_i][m]$.propose$(oper_i[m])$ is invoked only for the remaining objects, i.e., the objects m such that $m \in \{1,...,k\} \setminus L$. As in the algorithm of Figure 2, the important point is that a process invokes $AC[2r_i][ksc\_obj_x]$.propose() first on the set L of the objects output by the (k,ℓ)-SC object associated with the current round, and only after invoke it on the other objects.

Theorem 4 *When considering $\mathcal{ARW}_{n,n-1}[\emptyset]$, (k,ℓ)-UC and (k,ℓ)-SC have the same computational power:* (a) *a wait-free (k,ℓ)-UC algorithm can be implemented in $\mathcal{ARW}_{n,n-1}[(k,\ell)\text{-}SC]$, and* (b) *a wait-free (k,ℓ)-SC object can be built in $\mathcal{ARW}_{n,n-1}[(k,\ell)\text{-}UC$*

This theorem shows that (k,ℓ)-SC objects are both necessary and sufficient to ensure that at least ℓ objects always progress in a set of k objects. Let us remark that this is independent from the fact that the implementation of the k-universal construction is non-blocking or wait-free (going from non-blocking to wait-freedom requires the addition of a helping mechanism, but does not require additional computational power).

4.4 Relating $(k,k-p)$-SC and $(p+1)$-SA (i.e., $(p+1)$-SC) for $0 \le p \le k-1$: a Hierarchy

As indicated in the Introduction and shown in [1], k-set agreement (k-SA), which allows the processes to decide at most k different values from the proposed values, and k-SC are equivalent in $\mathcal{ARW}_{n,n-1}[\emptyset]$. This equivalence is denoted "$\equiv$" in Figure 3.

$(k,1)$ **and** (k,k)**-simultaneous consensus** It follows from its definition that $(k,1)$-SC is k-simultaneous consensus (i.e., equivalent to k-SA). At the other "extreme" case, (k,k)-SC is consensus as shown below.

- To solve consensus from (k,k)-SC, each process proposes its consensus input value in the first entry of its size k input vector, and decides the output in the first entry of the result vector.
- To solve (k,k)-SC from consensus, a vector of k consensus instances is used. A process proposes a value to each consensus instance, and the processes decide the same vector of size k, whose x-th entry contains a value proposed by a process to the x-th consensus instance.

In (k,k)-SC, a process always obtains a vector of size k (one entry per underlying consensus instance). It follows that, from an intuitive point of view, (k,k)-SC behaves as $(1,1)$-SC where a proposed value is a size k vector, i.e., (k,k)-SC behaves as consensus.

From $(k,k-p)$-simultaneous consensus to $(p+1)$-set agreement (i.e., $(p+1)$-simultaneous consensus) Let v_i be the value proposed by p_i to the $(1+p)$-set agreement; p_i proposes the size k vector $[v_i,...,v_i]$ to $(k,k-p)$-SC. Then it decides the maximal value of the $k-p$ outputs it obtains from $(k,k-p)$-SC. Hence, for any process p_i, at most p values among the k values proposed are greater than the value decided by p_i. It follows that at most $p+1$ values are decided, i.e., the values decided by the processes solves $(1+p)$-set agreement.

As $(1+p)$-set agreement is equivalent to $(1+p)$-SC in the basic read/write model, it follows that $(1+p)$-SC can be implemented in $\mathcal{ARW}_{n,n-1}[\emptyset]$ enriched with $(k,k-p)$-SC objects.

[4]It is possible to express (k,ℓ)-UC as an object accessed by appropriate operations. This is not done here because such an object formulation would be complicated without providing us with more insight on the question we are interested in.

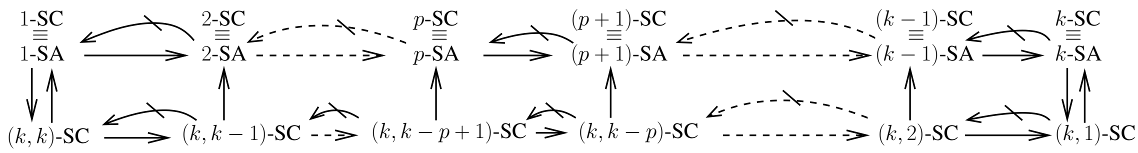

Figure 3: Relations linking $(p + 1)$-SA and $(k, k - p)$-SC, for $0 \leq p \leq k - 1$

These relations are summarized in Figure 3 which captures the relations linking the computational power of the kSA, k-SC and (k, ℓ)-SC objects. $A \rightarrow B$ means that B can be implemented in $\mathcal{ARW}_{n,n-1}[\emptyset]$ enriched with A, while $A \nrightarrow B$ means that B cannot be implemented in $\mathcal{ARW}_{n,n-1}[\emptyset]$ enriched with A. (Transitive possibility or impossibility reductions are not indicated.)

5 Conclusion

Our main objective was to build a universal construction for any set of k objects, each defined by a sequential specification, where at least ℓ of these k objects are guaranteed to progress forever. To that end, we have introduced a new object type, called (k, ℓ)-simultaneous consensus ((k, ℓ)-SC), and have shown that this object is both *necessary and sufficient* (hence optimal and universal) when one has to implement such a universal construction. We have related the notions of obstruction-freedom, non-blocking, and contention-awareness for the implementation of k-universal constructions. The paper has also introduced a general notion of *algorithm generosity*, which captures a property implicitly addressed in other contexts. More specifically, we have presented the following suite of constructions:

- A simple obstruction-free $(1, 1)$-universal construction based on atomic registers only (Appendix F).
- A contention-aware construction, based on k-SC objects and atomic registers, which is both obstruction-free (k, k)-universal and wait-free k-universal (Section 3).
- A contention-aware (k, ℓ)-universal construction based on (k, ℓ)-SC objects which is both obstruction-free (k, k)-universal and wait-free (k, ℓ)-universal (Section 4).

Finally, elements for a theory of (k, ℓ)-universality are presented in Appendix G.

References

[1] Afek Y., Gafni E., Rajsbaum S., Raynal M., and Travers C., The k-simultaneous consensus problem. *Distributed Computing*, 22(3):185-195, 2010.

[2] Anderson J.H. and Moir M., Universal constructions for large objects. *IEEE Transactions of Parallel and Distributed Systems*, 10(12):1317-1332, 1999.

[3] Attiya H., Guerraoui R., Hendler D., and Kutnetsov P., The complexity of obstruction-free implementations. *Journal of the ACM*, 56(4), Article 24, 33 pages, 2009.

[4] Attiya H. and Welch J.L., *Distributed computing: fundamentals, simulations and advanced topics, (2nd Edition)*, Wiley-Interscience, 414 pages, 2004 (ISBN 0-471-45324-2).

[5] Borowsky E. and Gafni E., Generalized FLP impossibility results for t-resilient asynchronous computations. *Proc. 25th ACM Symposium on Theory of Computing (STOC'93)*, ACM Press, pp. 91-100, 1993.

[6] Bouzid Z. and Travers C., Simultaneous consensus is harder than set agreement in message-passing. *Proc. 33rd Int'l IEEE Conference on Distributed Computing Systems (ICDCS'13)*, IEEE Press, pp. 611-620, 2013.

[7] Chaudhuri S., More choices allow more faults: set consensus problems in totally asynchronous systems. *Information and Computation*, 105(1):132-158, 1993.

[8] Chuong Ph., Ellen F. and Ramachandran V., A Universal construction for wait-free transaction friendly data structures. *Proc. 22th Int'l ACM Symposium on Parallelism in Algorithms and Architectures (SPAA'10)*, ACM Press, pp. 335-344, 2010.

[9] Crain T., Imbs D., and Raynal M., Towards a universal construction for transaction-based multiprocess programs. *Theoretical Computer Science*, 496:154-169, 2013.

[10] Ellen F., Fatourou P., Kosmas E., Milani A., and Travers C., Universal construction that ensure disjoint-access parallelism and wait-freedom. *Proc. 31th ACM Symposium on Principles of Distributed Computing (PODC)*, ACM Press, pp. 115-124, 2012.

[11] Fatourou P. and Kallimanis N.D., A highly-efficient wait-free universal construction. *Proc. 23th ACM Symposium on Parallel Algorithms and Architectures (SPAA)*, ACM Press, pp. 325-334, 2012.

[12] Fischer M.J., Lynch N.A., and Paterson M.S., Impossibility of distributed consensus with one faulty process. *Journal of the ACM*, 32(2):374-382, 1985.

[13] Gafni E., Round-by-round fault detectors: unifying synchrony and asynchrony. *Proc. 17th ACM Symposium on Principles of Distributed Computing (PODC)*, ACM Press, pp. 143-152, 1998.

[14] Gafni E. and Guerraoui R., Generalizing universality. *Proc. 22nd Int'l Conference on Concurrency Theory (CONCUR'11)*, Springer, LNCS 6901, pp. 17-27, 2011.

[15] Guerraoui R., Kapalka M., and Kouznetsov P., The weakest failure detectors to boost obstruction-freedom. *Distributed Computing* 20(6):415-433, 2008.

[16] Guerraoui R. and Lynch N.A., A general characterization of indulgence. *ACM Transactions on Autonomous and Adaptive Systems*, 3(4), Article 20, 2008.

[17] Herlihy M.P., Wait-free synchronization. *ACM Transactions on Programming Languages and Systems*, 13(1):124-149, 1991.

[18] Herlihy M.P., Luchangco V., and Moir M., Obstruction-free synchronization: double-ended queues as an example. *Proc. 23th Int'l IEEE Conference on Distributed Computing Systems (ICDCS'03)*, IEEE Press, pp. 522-529, 2003.

[19] Herlihy M., Luchangco V., Moir M., and Scherer III W.M., Software transactional memory for dynamic-sized data structures. *Proc. 22nd Int'l ACM Symposium on Principles of Distributed Computing (PODC'03)*, ACM Press, pp. 92-101, 2003.

[20] Herlihy M.P. and Moss J.E.B., Transactional memory: architectural support for lock-free data structures. *Proc. 20th ACM Int'l Symposium on Computer Architecture (ISCA'93)*, ACM Press, pp. 289-300, 1993.

[21] Herlihy M.P. and Shavit N., The topological structure of asynchronous computability. *Journal of the ACM*, 46(6):858-923, 1999.

[22] Herlihy M.P. and Wing J.M, Linearizability: a correctness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems*, 12(3):463-492, 1990.

[23] Lamport L., On inter-process communications, Part I: Basic formalism. *Distributed Computing*, 1(2): 77–85, 1986.

[24] Loui M. and Abu-Amara H., Memory requirements for agreement among unreliable asynchronous processes. *Advances in Computing Research*, 4:163-183, JAI Press, 1987.

[25] Lynch N.A., *Distributed algorithms*. Morgan Kaufmann, 872 pages, 1996.

[26] Luchangco V., Moir M., and Shavit N., On the Uncontended complexity of consensus. *Proc. 17th Int'l Symposium on Distributed Computing (DISC'03)*, Springer LNCS 2848, 45-59, 2003.

[27] Merritt M. and Taubenfeld G., Resilient consensus for infinitely many processes. *Proc. 17th Int'l Symposium on Distributed Computing (DISC'03)*, Springer LNCS 2848, 1-15, 2003.

[28] Raynal M., *Concurrent programming: algorithms, principles, and foundations*. Springer, 515 pages, 2013 (ISBN 978-3-642-32026-2).

[29] Raynal M. and Stainer J., Simultaneous consensus vs set agreement: a message-passing-sensitive hierarchy of agreement problems. *Proc. 20th Int'l Colloquium on Structural Information and Communication Complexity (SIROCCO 2013)*, Springer LNCS 8179, pp. 298-309, 2013.

[30] Saks M. and Zaharoglou F., Wait-free k-set agreement is impossible: the topology of public knowledge. *SIAM Journal on Computing*, 29(5):1449-1483, 2000.

[31] Shavit N. and Touitou D., Software transactional memory. *Distributed Computing*, 10(2):99-116, 1997.

[32] Taubenfeld G., Contention-sensitive data structure and algorithms. *Proc. 23rd Int'l Symposium on Distributed Computing (DISC'09)*, Springer LNCS 5805, pp. 157-171, 2009.

A Gafni and Guerraoui's Non-blocking k-Universal Construction

A.1 Gafni and Guerraoui's construction

This section presents Gafni and Guerraoui's generalized non-blocking k-universal construction introduced in [14], and denoted GG in the following. To make reading easier, we use the same variable names as in the construction presented in Figure 1 for local and shared objects that have the same meaning in both constructions. The objects considered in GG are deterministic state machines, and "operations" are accordingly called "commands".

Principle The algorithm GG is based on local replication, namely, the only shared objects are the control objects $kSC[1..]$ and $AC[1..][1..k]$. Each process p_i manages a copy of every state machine m, denoted $machine_i[m]$, which contains the last state of machine m as known by p_i. The invocation by p_i of $machine_i[m]$.execute(c) applies the command c to its local copy of machine m.

As explained in [14], the use of a naive strategy to update local copies of states machines, makes possible the following bad scenario. During a round r, a process p_1 executes a command $c1$ on its copy of machine $m1$, while a process p_2 executes a command $c2$ on a different machine $m2$. Then, during round $r+1$, p_1 executes a command $c2'$ on the machine $m2$ without having executed first $c2$ on its copy of $m2$. This bad behavior is prevented from occurring in [14] by a combined used of adopt-commit objects and an appropriate marking mechanism. When a process p_i applies a command c to its local copy of a machine m, it has necessarily received the pair $(commit, c)$ from the adopt-commit object associated with the current round, and consequently the other processes have received $(commit, c)$ or $(adopt, c)$. The process p_i attaches then to its next command for machine m, namely $oper_i[m]$, the indication that $oper_i[m]$ has to be applied to m after c, so that no process executes $oper_i[m]$ without having previously executed c.

```
r_i ← 0;
for each m ∈ {1, ..., k} do
      machine_i[m] ← initial state of the state machine m; oper_i[m] ← my_list_i[m].first()
end for.

repeat forever
(1)  r_i ← r_i + 1;
(2)  (ksc_obj, ksc_op) ← kSC[r_i].propose(oper_i[1..k]);
(3)  (tag_i[ksc_obj], ac_op_i[ksc_obj]) ← AC[r_i][ksc_obj].propose(ksc_op);
(4)  for each m ∈ {1, ..., k} \ {ksc_obj} do (tag_i[m], ac_op_i[m]) ← AC[r_i][m].propose(oper_i[m]) end for;
(5)  for each m ∈ {1, ..., k} do
(6)            if (ac_op_i[m] is marked "to_be_executed_after" oper_i[m])
(7)               then machine_i[m].execute(oper_i[m])
(8)            end if;
(9)            if (tag_i[m] = adopt)
(10)              then oper_i[m] ← ac_op_i[m]
(11)              else  machine_i[m].execute(ac_op_i[m]);                      % tag_i[m] = commit %
(12)                     if ac_op_i[m] = my_list_i[m].current()
(13)                        then oper_i[m] ← my_list_i[m].next()
(14)                        else  oper_i[m] ← my_list_i[m].current()
(15)                     end if;
(16)                     mark oper_i[m] "to_be_executed_after" ac_op_i[m]
(17)           end if
(18) end for
end repeat.
```

Figure 4: Gafni-Guerraoui's generalized universality non-blocking algorithm (code of p_i) [14]

Algorithm As before, $my\_list_i[m]$ defines the list of commands that p_i wants to apply to the machine m. Moreover, $my\_list_i[m]$.first() sets the read head to point to the first element of this list and returns its value; $my\_list_i[m]$.current() returns the command under the read head; finally, $my\_list_i[m]$.next() advances the read head before returning the command pointed to by the read head.

The algorithm is described in Figure 4. as the algorithm of Figure 1, it is round-based and has the same first four lines. When a process p_i enters a new asynchronous round (line 1), it first executes line 2-4, which are the lines involving the k-simultaneous consensus object and the adopt-commit object associated with the current round r.

After the execution of these lines, for $1 \leq m \leq k$, $(tag_i[m], ac\_op_i[m])$ contains the command that p_i has to consider for the machine m. For each of them it does the following. First, if $ac\_op_i[m]$ is marked "to be executed after" $oper_i[m]$, p_i applies

$oper_i[m]$ to $machine_i[m]$ (lines 6-8). Then, if $tag_i[m] = adopt$, p_i adopts $ac\_op_i[m]$ as its next proposal for machine m (lines 9-10). Otherwise, $tag_i[m] = commit$. In this case p_i first applies $ac\_op_i[m]$ to its local copy of the machine m (line 11). Then, if $ac\_op_i[m]$ was a command it has issued, p_i computes its next proposal $oper_i[m]$ for the machine m (lines 12-15). Finally, to prevent the bad behavior previously described, it attaches to $oper_i[m]$ the fact that this command cannot be applied to any copy of the machine m before the command $ac\_op_i[m]$ (line 16).

A.2 Discussion: Gafni-Guerraoui's construction revisited

The GG algorithm has two main drawbacks. First, it does not prevent a process from executing twice the same command on a given machine. Second, it is possible that, when a state machine stops progressing, it stops in different states at different processes. While the first issue can be easily fixed (see below), the second seems more difficult to fix.

Let us consider the following execution of the GG algorithm (Figure 4). During some round r, a process p_i applies a command c to its local copy of the machine m (hence, p_i obtained $(commit, c)$ from $AC[r][m]$, and each other process has obtained either $(commit, c)$ or $(adopt, c)$). It follows from line 16 that p_i marks its next command on m ($c' = oper_i[m]$) "to be executed after c". Let us consider now two distinct scenarios for the round $r + 1$.

Scenario 1. It is possible that all the processes, except p_i, have received $(adopt, c)$ during the round r and propose c to $AC[r + 1][m]$. Moreover, according to the specification of an adopt-commit object, nothing prevent $AC[r + 1][m]$ from outputting $(commit, c)$ at all the processes. In this case p_i will execute the command c twice on $machine_i[m]$. This erroneous behavior can be easily fixed by adding the following filtering after line 8:

if ($oper_i[m]$ is marked "to_be_executed_after"$ac\_op_i[m]$)
  then do not execute the lines 9-17
end if.

This filtering amounts to check if the command $ac\_op_i[m]$ has already been locally executed. The fact that $ac\_op_i[m]$ has been previously committed is encoded in $oper_i[m]$ by the marking mechanism.

Scenario 2. Let us again consider the round $r + 1$, and consider the possible case where the pair $(m, -)$ is not output by $kSC[r + 1]$ (let us remember that $kSC[r + 1]$ outputs one pair per process and globally at least one and at most k pairs). According to the specification of $AC[r + 1][m]$, it is possible to have $(tag_j[m], ac\_op_j[m]) = (adopt, c)$ at any process $p_j \neq p_i$, and $(tag_i[m], ac\_op_i[m]) = (adopt, c')$ where c' is the new command that p_i wants to apply to the machine m. Hence, as far as m is concerned, all the processes execute the lines 9-10, and we are in the same configuration as at the end of round r. It follows that this can repeat forever. If it is the case, p_i has executed one more command on its local copy of machine m than the other processes. This means that state machine m stops progressing in different states at distinct processes.

B Proofs of the Lemmas and Theorem of Section 3.4

To make this appendix self-contained, some definitions and explanations of Section 3.4 are repeated here.

Lemma 1 $\forall\, i, m$: $(op \in GSTATE[i][m]) \Rightarrow (\exists\, j : op \in my\_list_j[m])$ (i.e., if an operation op is applied to an object m, then op has been proposed by a process).

Proof Before being written into $GSTATE[i][m]$ (line 31), an operation op is first appended to m's local history for the first time at line 18. It follows from lines 2-4 that this operation was proposed to an adopt-commit object by some process p_j in $oper_j[m]$. If $oper_j[m]$ was updated in the initialization phase, at line 24 or line 27, it is an operation of $my\_list_j[m]$. If $oper_j[m]$ was updated at line 25, it was proposed to an adopt-commit object by another process p_x, and (by a simple induction) the previous reasoning shows that this operation belongs then to some $my\_list_z[m]$. $\Box_{Lemma\ 1}$

Lemma 2 $\forall\, i, j, m : (op \in my\_list_j[m]) \Rightarrow$ (op appears at most once in $GSTATE[i][m]$) (i.e., an operation is executed at most once).

Proof Suppose by contradiction that, at a given time and for an object m, a history $GSTATE[-][m]$ contains twice the same operation op. Let p_i be the first process that wrote such a history with op appearing twice in $GSTATE[i][m]$, and let τ be the time instant at which p_i does it. Since $GSTATE[i][m]$ is written only at line 31 with the content of $\ell\_hist_i[m]$, p_i necessarily stored before τ an history containing twice op in $\ell\_hist_i[m]$. As $\ell\_hist_i[m]$ is initially empty, it does not contain twice op in the initial state of p_i. Since $\ell\_hist_i[m]$ is updated only at line 7 or line 18, p_i sets it to a history containing twice op at one of these lines. According to the predicate of line 16, p_i cannot append op to $\ell\_hist_i[m]$ at line 18 if op already appears in that sequence. It follows that p_i updates $\ell\_hist_i[m]$ before τ at line 7 with one of the longest local histories of m which contains op twice. Consequently, when p_i read (non-atomically) $GSTATE$ at line 5, it retrieved that history from one of the $GSTATE[j][m]$, also before τ. But this contradicts the fact that no process writes a history containing op twice before τ. It follows that no history containing several times the same operation can ever be written into one of the registers $GSTATE[-][-]$. $\Box_{Lemma\ 2}$

The sequence $(op_r^m)_{r\geq 1}$ of committed operations According to the specification of the adopt-commit object, for any round r and any object m there is at most one operation returned with the tag $commit$ by the object $AC[r][m]$ to some processes. Let op_r^m denote this unique operation if at least one process obtains a pair with the tag $commit$, and let op_r^m be $\perp$ if all the pairs returned by $AC[r][m]$ contain the tag $adopt$.

From the sequence $(op_r^m)_{r\geq 1}$ to the notion of valid histories Considering an execution of the algorithm of Figure 1, the following lemmas show that, for any process p_i and any object m, all the sequences of operations appearing in $\ell\_hist_i[m]$ are finite prefixes of a unique valid sequence depending only on the sequence $(op_r^m)_{r\geq 1}$ of committed operations.

More precisely, given a sequence $(op_r^m)_{r\geq 1}$, a history $(vh_x^m)_{1\leq x\leq xmax}$ is *valid* if it is equal to a sequence $(op_r^m)_{1\leq r\leq R}$ from which the $\perp$ values and the repetitions have been removed. More formally, $(vh_x^m)_{1\leq x\leq xmax}$ is valid if there is a round number R and a strictly increasing function $\sigma : \{1,\ldots,xmax\} \to \{1,\ldots,R\}$ such that for all x in $\{1,\ldots,xmax\}$: (a) $vh_x^m = op_{\sigma(x)}^m$, (b) $vh_x^m \neq \perp$, (c) for all x in $\{1,\ldots,xmax-1\}$: $vh_x^m \neq vh_{x+1}^m$, and (d) the sets $\{vh_1^m,...,vh_{xmax}^m\}$ and $\{op_1^m,...,op_R^m\} \setminus \{\perp\}$ are equal.

Let us remark that this definition has two consequences: (i) the value of R for which item (d) is verified defines unambiguously the sequence $(vh_x^m)_{1\leq x\leq xmax}$ (and accordingly this sequence is denoted $VH^m(R)$ in the following), and (ii) for any two valid histories $(vh_x^m)_{1\leq x\leq xmax1}$ and $(vh_x^m)_{1\leq x\leq xmax2}$, one is a prefix of the other.

Lemma 3 For any process p_i and any object m, at any time the local history $\ell\_hist_i[m]$ is valid.

Proof Let us suppose by contradiction that a process p_j updates $\ell\_hist_j[m]$ with a sequence that is not valid. Let p_i be the first process that writes an invalid sequence (denoted s) into its variable $\ell\_hist_i[m]$. Let ρ be the round and τ the time at which it does it.

Since p_i is the first process that writes s into its local history $\ell\_hist_i[m]$, it cannot do it at line 7 (this would imply that p_i retrieved s in some $g\_state_i[j][m]$ obtained from its previous non-atomic read of $GSTATE$ –line 5– implying that a process p_j would have written s into its local history $\ell\_hist_j[m]$ before τ). Consequently p_i writes s into $\ell\_hist_i[m]$ at line 18. It follows that the adopt-commit object $AC[\rho][m]$ returned to p_i the pair $(commit, op)$ (where op is the last operation in s) at line 3 or 4 during round ρ, hence, $op_\rho^m = op$.

Let us remind that, by assumption, before p_i appended op to $\ell\_hist_i[m]$ at line 18 of round ρ, $\ell\_hist_i[m]$ was valid; let s' denote that history. Moreover, as p_i executes line 18 of round ρ, it fulfilled the condition of line 16, hence we have $op \notin s'$. Let R_1 be the smallest (resp. R_2 the largest) round number R such that $s' = VH^m(R)$. It follows from the previous observation that $R_2 < \rho$, and from the definition of R_1, that $op_{R_1}^m \neq \perp$ ($op_{R_1}^m$ is the last operation appearing in $VH^m(R_1) = VH^m(R_2)$). Let us remark that, since s' is valid while s is not, there is necessarily a round number r such that $R_2 < r < \rho$, $op_r^m \neq \perp$ and $s' = VH^m(R_2) \neq VH^m(r)$ (intuitively, p_i "missed" a committed operation). Let r_0 be the smallest round number verifying these conditions. According to this definition, $op_{r_0}^m \neq op_{R_1}^m$.

Let us first show that $op_{r_0}^m \notin VH^m(R_1) = VH^m(R_2)$. Suppose by contradiction that it exists a round $r_1 < R_2$ such that $op_{r_1}^m = op_{r_0}^m$ and consider a process p_j executing round r_1. The proof boils down to show that such a process p_j cannot propose $op_{r_1}^m = op_{r_0}^m$ to a $kSC[r]$ object with $r > r_1 + 1$ before τ, which entails that this operation cannot be committed during round r_0 and leads to a contradiction. If p_j commits $op_{r_1}^m = op_{r_0}^m$ during that round, then, after the execution of lines 16-28, it has op_{r_1} into its variable $\ell\_hist_i[m]$, has set its variable $oper_j[m]$ to a different operation and will never propose op_{r_1} further in the execution. If p_j adopts op_{r_1} during round r_1, then two cases are possible: (a) p_j returns from its invocation of $AC[r_1+1][m].\mathsf{propose}(-)$ before that any process, which has committed op_{r_1} during round r_1, invokes $kSC[r_1+1][m].\mathsf{propose}(-)$, or (b) one of the processes that committed op_{r_1} during round r_1, invokes $kSC[r_1+1][m].\mathsf{propose}(-)$ before p_j returns from its invocation of $AC[r_1+1][m].\mathsf{propose}(-)$. In the case (a), according to the validity properties of the k-simultaneous consensus and adopt-commit objects, p_j commits op_{r_1} during round r_1+1 and, as before, will not propose this operation further in the execution since it appears in its local history. In the case (b), one of the processes that committed op_{r_1} during round r_1 wrote an history containing it before p_j executes line 5 of round r_1+1. If this happens before τ, then both this history and the history of p_j are valid, thus p_j adopts that history that strictly contains its own local history. It follows that p_j executes lines 16-28 of round r_1+1 with an history containing op_{r_1} and consequently never proposes this operation further in the execution. This ends the proof of the fact that $op_{r_0}^m \notin VH^m(R_1) = VH^m(R_2)$.

From the previous remark, it follows that, before τ, p_i never retrieves any history $VH^m(r)$ with $r \geq r_0$ during its non-atomic read of $GSTATE$ (or it would have set its variable $\ell\_hist_i[m]$ to one of these histories at line 7 and never reset it to s', since these histories contain $VH^m(r_0)$, and are consequently strictly longer than s').

Let us consider the execution of round r_0 by p_i (since p_i reaches line 18 of round $\rho > r_0$, this occurs). Let us suppose that p_i obtains the pair $(commit, op^m_{r_0})$ from $AC[r_0][m]$. As, (a) before τ, the values of $\ell\_hist_i[m]$ are valid (hence they can only increase), and (b) $op^m_{r_0} \notin VH^m(R_2)$, it follows that p_i appends $op^m_{r_0}$ to $\ell\_hist_i[m]$ at line 18 of round r_0, contradicting the fact that, just before τ, $\ell\_hist_i[m] = s' = VH^m(R_2)$. Consequently, according to the definition of r_0 and the specification of the adopt-commit object, $AC[r_0][m]$ returns $(adopt, op^m_{r_0})$ to p_i.

During round r_0, since $op^m_{r_0} \neq \perp$, all the processes that do not crash before obtain one of the two pairs $(adopt, op^m_{r_0})$ or $(commit, op^m_{r_0})$ from $AC[r_0][m]$. Let $\mathcal{C}$ denote the ones that obtain $(commit, op^m_{r_0})$, and $\mathcal{A}$ the one that obtain $(adopt, op^m_{r_0})$. Among the processes of $\mathcal{A}$, some fulfills the condition of line 16 during round r_0, namely those which do not have $op^m_{r_0}$ in their local history. Let $\mathcal{A}_-$ denote this set of processes and let $\mathcal{A}_+$ be $\mathcal{A} \setminus \mathcal{A}_-$. As previously shown, p_i cannot have $op^m_{r_0}$ in $\ell\_hist_i[m]$ before τ; consequently $p_i \in \mathcal{A}_-$. Let μ be the first time at which a process of $\mathcal{C} \cup \mathcal{A}_+$ (the set of processes that have $op^m_{r_0}$ in their local histories at the end of round r_0) executes line 31 of round r_0. Let μ' be the first time at which one of these processes invokes $kSC[r_0+1][m]$.propose$(-)$ at round r_0+1. Let τ_i be the time at which p_i terminates its invocation of $AC[r_0+1][m]$.propose$(-)$, and τ'_i the time at which it terminates its read of line 5 during round r_0+1.

Let us remark that any process p_j of $\mathcal{A}_-$ (including p_i) starts round r_0+1 with $oper_j[m] = op^m_{r_0}$. It follows from the k-simultaneous consensus and adopt-commit specifications and the structure of the lines 2-4, that if $\tau_i < \mu'$ then p_i necessarily obtains the pair $(commit, op^m_{r_0})$ from $AC[r_0+1][m]$. As this happens before τ, $op^m_{r_0} \notin \ell\_hist_i[m]$ when p_i checks the condition of line 16, and it consequently appends op_{r_0} to $\ell\_hist_i[m]$ at line 18 of round r_0+1. This is contradicts the fact that $s' = VH^m(R_2)$, except for the case $r_0+1 = \rho$. But, for $r_0+1 = \rho$, we should have $op^m_{r_0} = op^m_{\rho} = op$, and, by definition of r_0, s would be valid, which contradicts the fact that (due to the definition of s) it is not.

The only remaining case is thus $\mu' < \tau_i$, but since $\mu < \mu'$ and $\tau_i < \tau'_i$, it follows that $\mu < \tau'_i$ which implies that p_i obtains a valid history containing op_{r_0} during its read of $GSTATE$ at round r_0+1 and consequently updates $\ell\_hist_i[m]$ to one of these histories at line 7, thus before τ. This leads to a contradiction which concludes the proof of the lemma. $\Box_{Lemma\ 3}$

The execution on an object m of an operation op, issued by a process p_i, starts when the process p_i proposes op to a k-simultaneous consensus object $kSC[-][m]$ for the first time (i.e., when p_i makes op public), and terminates when a set res including $(m, op, output[m])$ is returned by p_i at line 10 or line 31. The next lemma shows that any execution is linearizable.

Lemma 4 The execution of an operation op issued by a process p_i on an object m can be linearized at the first time at which a process p_j writes into $GSTATE[j][m]$ a local history $\ell\_hist_j[m]$ such that $op \in \ell\_hist_j[m]$.

Proof Let op be an operation applied on an object m and p_i be the process such that $op \in my\_list_i[m]$. Let us first show that op cannot appear in the local history $\ell\_hist_j[m]$ before being proposed by p_i to one of the k-simultaneous consensus objects $kSC[-][m]$. Let p_j be the first process that adds op to its local history $\ell\_hist_j[m]$ and τ the time at which this occurs. It follows that time τ cannot occur at line 7, but occurs when p_j executes line 18 when it appends op to $\ell\_hist_j[m]$ during some round r. Process p_j consequently obtained the pair $(commit, op)$ from the adopt-commit object $AC[r][m]$ at line 3 or line 3 of round r. According to the validity properties of k-simultaneous consensus and adopt-commit objects and to the structure of the lines 2 to 4, it follows that a process proposed op to $kSC[r][m]$ before τ.

There are two ways for a process to propose op to $kSC[r][m]$: either (a) it adopted it at line 25 of round $r-1$ (if $r > 1$) or (b) the process is p_i, $op \in my\_list_i[m]$, and p_i wrote op into $oper_i$ at line 24 or line 27 of round $r-1$ (if $r > 1$), or during initialization (if $r = 1$). With the same reasoning as in the previous paragraph, case (a) implies that a process proposed op to $kSC[r-1][m]$ before τ. This can be explained by case (a) at round $r-2$ only if $r > 2$, or by case (b) at round $r-2$. By iterating this reasoning, in the worst case until reaching round 1, it comes that in any case (b) happened, and that p_i necessarily proposed op to one of the $kSC[-][m]$ objects before τ. Consequently, no process p_j has op in $\ell\_hist_j[m]$ before p_i proposed it to one of the $kSC[-][m]$ objects, thus the linearization point of op is after p_i has made public the operation op.

On the other hand, if it terminates, the operation op issued by p_i ends at lines 10 or 31 after that p_i computed an output for op. It can do it only at lines 9 or 20, and, in both cases, thanks to line 8 or lines 18-19, this happens only when op appears in $\ell\_hist_i[m]$. This implies that p_i either obtained a history containing op at line 5 of the same round, or writes a history containing op in $GSTATE[i][m]$ at line 30 of the same round before executing line 31, which proves that the linearization point of op is before op terminates at p_i (if it ever terminates).

Finally, according to Lemma 3, all the processes construct the same history of operations on m. Since the results locally returned are appropriately computed with compute_output() on the right prefix of the local history of m, the sequential specification of the object m is satisfied. This concludes the fact that there is a linearization of the sequence of operations applied on any object m. As any object m is linearizable, and as linearizability is a local property [22], it follows that the execution is linearizable, which ends the proof of the lemma. $\Box_{Lemma\ 4}$

Lemma 5 $\forall\ r \geq 1$, there is a process p_i such that at least one operation op output by $kSC[r]$.propose() at p_i (line *2*) is such that the invocation of $AC[r][-]$.propose() by p_i returns $(commit, op)$ (line *3* or *4*).

Proof The proof is based on an observation presented in [14]. Let us first notice that, after it has received a pair (ksc_obj_1, ksc_op_1) from $kSC[r]$.propose() at line 2, a process p_{i1} invokes first $AC[r][ksc\_obj_1]$.propose(ksc_op_1) at line 3 before invoking $AC[r][ksc\_obj]$.prop at line 4 for any object $ksc\_obj \neq ksc\_obj_1$. If the invocation $AC[r][ksc\_obj_1]$.propose(ksc_op_1) issued by p_{i1} returns the pair $(commit, -)$, the lemma follows.

Hence, let us assume that the invocation by p_{i1} of $AC[r][ksc\_obj_1]$.propose(ksc_op_1) at line 3 returns the pair $(adopt, -)$. It follows from the "non-conflicting values" property of the adopt-commit object $AC[r][ksc\_obj_1]$, that a process p_{i2} has necessarily invoked $AC[r][ksc\_obj_1]$.propose(op'), with $op' \neq ksc\_op_1$, and this invocation was issued at line 4 (if both p_{i1} and p_{i2} had invoked $AC[r][ksc\_obj_1]$.propose() at line 3, they would have obtained the same pair from the object $kSC[r]$ at line 2, and consequently, p_{i2} could not prevent p_{i1} from obtaining $(commit, -)$ from the adopt-commit object $AC[r][ksc\_obj_1]$). It follows that p_{i2} starts line 4 before p_{i1} terminates line 3. The invocation by p_{i2} of $AC[r][-]$ at line 3 involved some object ksc_obj_2 obtained by p_{i2} from its invocation of $kSC[r]$.propose() at line 2 (as seen previously, we necessarily have $ksc\_obj_2 \neq ksc\_obj_1$).

If the invocation by p_{i2} of $AC[r][ksc\_obj_2]$.propose() returns $(commit, -)$, the lemma follows. Otherwise, due to the "non-conflicting values" property of adopt-commit, there is a process p_{i3} that prevented p_{i2} from obtaining $(commit, -)$ from its invocation of $AC[r][ksc\_obj_2]$.propose() at line 3. let us notice that $p_{i3} \neq p_{i1}$ (this follows from the observation that p_{i3} started line 4 before p_{i2} terminates line 3, which itself started line 4 before p_{i1} terminates line 3, hence p_{i3} started line 4 before p_{i1} terminates line 3). The execution pattern between p_{i2} and p_{i3} is then the same as the previous pattern between p_{i1} and p_{i2}. While this pattern can be reproduced between p_{i3} and another process p_{i4}, then between p_{i4} and p_{i5}, etc., its number of occurrences is necessarily bounded because the number of processes is bounded. It then follows that there is a process p_{ix} that obtains the pair $(commit, -)$ when it invokes $AC[r][ksc\_obj_{ix}]$.propose() at line 3 (where ksc_obj_{ix} is the object returned to p_{ix} by its invocation $kSC[r]$.propose() at line 2). $\Box_{Lemma\ 5}$

Lemma 6 There is at least one object on which an infinite number of operations are executed.

Proof This lemma follows from (a) the fact that an operation committed during some round at some process is eventually made globally visible in $GSTATE$ (lines 17, 18, and 30), (b) Lemma 5 (at every round an operation is committed at some process), and (c) the fact that the number of objects is bounded. $\Box_{Lemma\ 6}$

It follows from the previous lemma, and the fact that there is a bounded number of processes, that at least one process executes an infinite number of its operations on an object. Hence the following corollary.

Corollary 2 *The algorithm is non-blocking.*

Theorem 1 The algorithm of Figure 1 is a non-blocking linearizable $(k, 1)$-universal construction.

Proof The proof follows from the previous lemmas and corollary. $\Box_{Theorem\ 1}$

C Eliminating Full Object Histories

For each process p_i and object m, the universal construction uses a shared register $GSTATE[i][m]$ to remember the sequence of all the operations that have been successfully applied to object m, as currently known to p_i. We have chosen this implementation mainly due to its simplicity. While it is space inefficient, it can be improved as follows.

- Recall that we have assumed that all the operations are unique. This can be easily implemented locally, where each process attaches a unique (local) sequence number plus its id to each operation. The (local) sequence number attached can be the number of operations the process has invoked on the object so far. Now, instead of remembering (by each process) for each object m its full history, it is sufficient that each process p_i computes and remembers only the last state of m, denoted $\ell\_state_i[m]$, plus the sequence number of the last operation successfully applied to m by each process.
- As far as the function compute_output(op, h) used at line 9 and line 20 is concerned, we have the following, where $OUTPUT[1..n]$ is an array made up of one atomic register per process. Immediately after line 18, a process p_i executes the following statements, which replace lines 19-23.

$output_i[m] \leftarrow$ compute_output$(ac_op_i[m], \ell_state_i[m])$;
let p_j **be** the process that invoked $ac\_op_i[m]$;
if $(i = j)$ **then** lines 21-22
else $OUTPUT[j] \leftarrow output_i[m]$
end if.

Finally, when executed by a process p_j, line 9 is replaced by $output_j[m] \leftarrow OUTPUT[j]$.

It is easy to see that these statements implement a simple helping mechanism that allow processes, which invoke append() at line 18, to pre-compute the operation results for the processes that should invoke compute_output(op, h) at line 9. Consequently, the distributed universal construction can be easily modified to use this more space efficient representation instead of the "full history" representation.

D Proofs of the Theorems of Section 4

Theorem 2 The algorithm of Figure 2 is a non-blocking contention-aware $(k, 1)$-universal construction.

Proof The proof first shows that the modified code provides the same safety guarantees than the previous construction. Namely, for any m, if a process p_i terminates line N3 with $tag_i[m] = commit$, then any process p_j executing line N3 ends it with $ac\_op_j[m] = ac\_op_i[m]$. Let us remark that if p_i retrieves the pair $(commit, ac\_op_i[m])$ from $AC[2r_i - 1][m]$ at line N1, it follows from the property of the adopt-commit object that any other process p_j executing this line finishes it with $ac\_op_j[m] = ac\_op_i[m]$. Consequently all processes executing lines 2M to 4M propose only this value to the k-simultaneous consensus object at line 2M or to the $AC[2r_i][m]$ object at line 4M. Moreover according to the validity of the k-simultaneous consensus object, if a process retrieves a pair (m, ksc_op) from the k-simultaneous consensus of line 2M then $ksc\_op = ac\_op_i[m]$, thus $ac\_op_i[m]$ is the only value that can be proposed to $AC[2r_i][m]$ at line 3. It follows that if a process retrieves a pair $(commit, op)$ from $AC[2r_i - 1][m]$ then any process p_j that executes lines 2M to 4M finishes line 4M with $ac\_op_j[m] = op$, while, thanks to the agreement property of $AC[2r_i - 1][m]$, any process p_h that do not execute lines 2M to 4M also ends line N3 with $ac\_op_h[m] = op$. Additionally, if a process obtains a pair $(commit, op)$ from $AC[2r_i][m]$ while all processes obtain $(adopt, -)$ from $AC[2r_i - 1][m]$, then each process p_j executes lines 2M to 4M and thus, according to the agreement property of $AC[2r_i][m]$, obtains a pair $(-, op)$ from it and finishes line 4M with $ac\_op_j[m] = op$.

Moreover, the progress property verified by the previous construction is preserved: for any m, if a process p_i which starts line N1 with $oper_i[m] = op$, finishes the execution of line N3 before any process p_j with $oper_j[m] \neq op$ executes line N1, then p_i ends line N3 with $tag_i[m] = commit$ and $ac\_op_i[m] = op$. This comes directly from the validity properties of the k-simultaneous consensus and adopt-commit objects.

Finally, if a process executes alone, the k-simultaneous consensus object is not used and all the objects progress, while, in case of contention, as before, at least one object progresses (the first part comes from the validity property of the $AC[2r_i - 1][-]$ objects and the condition stated at line N2; the second part comes from Lemma 5).

Thanks to the previous observations, the lemmas of Theorem 1 hold with the modified code, which ends this proof. $\Box_{Theorem\ 2}$

Theorem 3 When replacing the lines *24* and *27* by lines L1-L5, the algorithms of Figure 1 and Figure 2 are wait-free linearizable $(k, 1)$-universal constructions.

Proof Let us first observe that the lines 1-N3 of Figure 2 do not access the local variables $my\_op_i[m]$, and consequently have no impact on the lines 24 and 27 replaced by the new lines L1-L5.

An increase of a local history $\ell\_hist_i[m]$ is *direct* if it occurs at line 18, and *indirect* if it occurs at line 7. Let us observe that a direct increase adds one operation to a local history. Moreover, all increases are caused by direct increases, which can then be propagated by indirect increases.

All the time instants considered in this proof are time instants after which all faulty processes have crashed. Let m be an object which progresses forever. Let p_j be a correct process such that the last operation it has written in $LAST\_OP[j, m]$ is never executed. Let $op(j, m)$ denotes this operation. The proof is by contradiction.

Let r be a round such that (a) $op(j, m)$ has been written in $LAST\_OP[j, m]$, and (b) there is a direct increase such that there is a process p_i such that $|\ell\_hist_i[m]| \mod n + 1 = j$. Let us observe that, as the object m progresses forever and all increases are due to direct increases, both such a round r and process p_i do exist. Moreover, as it is a direct increase, p_i executed line 18 from which it follows that it executes line 24 of round r. Hence, p_i executes the new code L1-L5 of the lines 24 and we necessarily have $oper_i[m] = op(j, m)$.

If, during round r, all processes execute the new code L1-L5 of lines 24 or 27, they all starts the next round $r+1$ with $oper_i[m] = op(j,m)$, and consequently $op(j,m)$ will be committed during round $r+1$. In this case, $op(j,m)$ will be executed, contradicting the initial assumption. Hence, let us assume that a process p_h executes line 25 during round r. We have $oper_h[m] = ac\_op_h[m]$, where $ac\_op_h[m] = op$ is the operation committed by p_i at round r. Let us observe that we have then necessarily $|\ell\_hist_h[m]| = |\ell\_hist_i[m]| - 1$ (p_i has added op to $\ell\_hist_i[m]$ while p_h has not yet done it). We consider two cases.

- Process p_h terminates line N3 before p_i (or any other process which behaves as p_i) starts line N1. In this case, p_h terminates line N3 with the pair $(tag_i[m], ac\_op_i[m]) = (commit, op)$, and consequently adds op to $\ell\_hist_h[m]$. We have now $|\ell\_hist_h[m]| = |\ell\_hist_i[m]|$, and all the processes p_x that proceed to the round $r+2$, are such that $oper_x[m] = op(j,m)$. It follows that $op(j,m)$ will be committed during round $r+2$, which contradicts our assumption.
- Process p_i (or a process that, during round r, behaves as p_i, i.e., which has committed an operation on m –necessarily op–) starts line N1 before p_h (or a process which behaves as p_h) has terminated line N3. It follows that p_h terminates line N3 with either $ac\_op_h[m] = op(j,m)$ or $ac\_op_h[m] = op$ (the operation stored in $oper_h[m]$ and committed by p_i at round r).

  In this case, p_i has made public $\ell\_hist_i[m]$ (line 30) before p_h reads $GSTATE[i][m]$ (line 5). Hence, p_h reads the local history $\ell\_hist_i[m]$, and consequently $\ell\_hist_h[m]$ contains $\ell\_hist_i[m]$. Moreover, we also have $op \in \ell\_hist_h[m]$ when p_i executes the body of the loop of line 15 for object m. We consider two sub-cases.

  - $\ell\_hist_h[m] = \ell\_hist_i[m]$.
    * If $ac\_op_h[m] = op$: then $ac\_op_h[m] \in \ell\_hist_h[m]$, and p_h executes the new code L1-L5 of line 27. As $\ell\_hist_h[m] = \ell\_hist_i[m]$, we consequently have $oper_h[m] = op(j,m)$, from which it follows that every process p_x start the next round $r+2$ with $oper_x[m] = op(j,m)$; $op(j,m)$ is then committed during the next round, which contradicts our assumption.
    * If $ac\_op_h[m] = op(j,m)$ and the associated tag is $adopt$: p_h executes line 25, and we have $oper_h[m] = op(j,m)$. If $ac\_op_h[m] = op(j,m)$ and the associated tag is $commit$: the processes commit $op(j,m)$. In both case, $op(j,m)$is committed (at the current round or the next one), which contradicts the initial assumption.
  - $\ell\_hist_i[m]$ is a strict prefix of $\ell\_hist_h[m]$. In this case, p_h does not participate in the commitment of the operation on m that follows op in $\ell\_hist_h[m]$. It perceived it from an indirect increase of $\ell\_hist_h[m]$.

If follows from the previous reasoning that the initial assumption (namely, $op(j,m)$ is never committed) is contradicted. Consequently $op(j,m)$ is committed. As this is true for any correct process p_j and any object m that progresses forever, it follows that any correct process executes an infinite number of operations on any object that progresses forever. $\Box_{Theorem\ 3}$

Theorem 4 When considering $\mathcal{ARW}_{n,n-1}[\emptyset]$, (k,ℓ)-UC and (k,ℓ)-SC have the same computational power: *(a)* a wait-free (k,ℓ)-UC algorithm can be implemented in $\mathcal{ARW}_{n,n-1}[(k,\ell)\text{-}SC]$, and *(b)* a wait-free (k,ℓ)-SC object can be built in $\mathcal{ARW}_{n,n-1}[(k,\ell)\text{-}UC$

Proof Proof of (a). The proof that a (k,ℓ)-UC algorithm can be implemented in $\mathcal{ARW}_{n,n-1}[(k,\ell)\text{-}SC]$ amounts to show that (k,ℓ)-SC allows at least ℓ objects to progress forever. If during a given round one of the processes does not verify the condition of line N2, as noticed in the proof of Theorem 2, all the objects progress. If all the processes execute lines 2M to 4M, then the reasoning of Lemma 5 holds and at least one process obtains only $commit$ tags at line 3 from the ℓ adopt-commit objects associated with the ℓ objects for which it obtained operations from the (k,ℓ)-SC object associated with the corresponding round. Consequently, during any round, at least ℓ objects progress.

Proof of (b). To prove that a (k,ℓ)-SC object can be built in $\mathcal{ARW}_{n,n-1}[(k,\ell)\text{-}UC]$, let us consider an algorithm (k,ℓ)-UC where the k concurrent objects it is instantiated with are atomic read/write registers. Moreover, on each object m, a process p_i issues a write operation followed by read operations. When a process p_i wants to propose to the (k,ℓ)-SC object the vector $[v_i^1, ..., v_i^k]$, it invokes for each $m \in \{1,...,k\}$, the operation $write(v_i^m)$ on the corresponding object m. Due to the (k,ℓ)-UC algorithm, each process sees at least ℓ objects progress. As soon as a process p_i sees that ℓ objects have progressed, it returns an output vector of size k containing the ℓ values written in these objects, and $\perp$ at each of the $k-\ell$ remaining entries. Hence, a process p_i returns a vector of size k with exactly ℓ non-$\perp$ entries. Moreover, it follows from the (k,ℓ)-UC algorithm that, the processes see the same sequence of operations on each object. Hence, if p_i returns $v \neq \perp$ and p_j returns $v' \neq \perp$ for the same entry m of their output arrays, these values have been written by the same write operation, and are consequently such that $v = v'$, which concludes the proof. $\Box_{Theorem\ 4}$

E Contention Awareness: Reducing the Number of Uses of k-SC Objects

As announced in Section 4.1, it is possible to reduce the number of uses of the underlying k-SC synchronization objects. This is obtained by replacing the lines N1 until N3 in Figure 2 by the lines as described in Figure 5. There is one modified line (N2M) and three new lines (NN1, NN2, and NN3).

```
(N1)   for each m ∈ {1,...,k} do (tag_i[m], ac_op_i[m]) ← AC[2r_i − 1][m].propose(oper_i[m]) end for;
(N2M)  if (∀ m ∈ {1,...,k} : tag_i[m] = adopt)                                  % ∀ m replaces ∃ m%
(2M)     then (ksc_obj, ksc_op) ← kSC[r_i].propose(ac_op_i[1..k]);
(3)           (tag_i[ksc_obj], ac_op_i[ksc_obj]) ← AC[2r_i][ksc_obj].propose(ksc_op);
(4M)          for each m ∈ {1,...,k} \ {ksc_obj} do (tag_i[m], ac_op_i[m]) ← AC[2r_i][m].propose(ac_op_i[m]) end for
(NN1)    else for each m ∈ {1,...,k} do
(NN2)           if (tag_i[m] = adopt) then (tag_i[m], ac_op_i[m]) ← AC[2r_i][m].propose(ac_op_i[m]) end if
(NN3)         end for
(N3)   end if.
```

Figure 5: Efficient Contention-aware Non-Blocking $(k, 1)$-Universal Construction (code for p_i)

More precisely, if after it has used the adopt-commit objects $AC[2r_i - 1][m]$, for each constructed object m, p_i has received only tags $adopt$ (modified line N2M), it executes the lines 2M, 3, and 4M, as in basic contention aware construction of Figure 2. Differently, if it has received the tag $commit$ for at least one constructed object, it invokes $AC[2r][m]$ for all the objects m for which it has received the tag $adopt$ (new lines NN1-NN3).

F Obstruction-Free Construction Based on Read/write Registers Only

A remark on obstruction-freedom and generosity The reader can check that the three above constructions (Sections 3.3, 4.2 and 4.3), are obstruction-free (k, k)-universal constructions. More precisely, (1) at least one (or ℓ) objects are guaranteed to always progress under contention, and (2) it is guaranteed that a process will be able to complete its pending operation in a finite number of steps, if all the other processes "hold still" long enough. It follows from (2) that, if once in a while all the processes except one "hold still" long enough, then all the k objects are guaranteed to always progress.

```
my_op_i ← my_list_i.next(); oper_i ← my_op_i.

repeat forever
(1)  r_i ← r_i + 1;
(3)  (tag_i, ac_op_i) ← AC[r_i].propose(oper_i);
(5)  for each j ∈ {1,...,n} do g_state_i[j] ← GSTATE[j] end for;
(7)  ℓ_hist_i ← longest history in g_state_i[1..n] containing ℓ_hist_i;
(8)  if (my_op_i ∈ ℓ_hist_i)                                   % my operation was completed %
(9)     then output_i ← compute_output(my_op_i, ℓ_hist_i);
(10)         return (my_op_i, output_i) to the upper layer;
(11)         my_op_i ← my_list.next()
(12) end if;
(14) res_i ← ⊥;
(16) if (ac_op_i ∉ ℓ_hist_i)                                   % operation was not completed %
(17)         then if (tag_i = commit)                            % complete the operation %
(18)                 then ℓ_hist_i ← ℓ_hist_i.append(ac_op_i);
(19)                      if (ac_op_i = my_op_i)               % my operation was completed %
(20)                         then output_i ← compute_output(my_op_i, ℓ_hist_i);
(21)                              res ← (my_op_i, output_i);
(22)                              my_op_i ← my_list.next()
(23)                      end if;
(24)                      oper_i ← my_op_i
(25)                 else oper_i ← ac_op_i                                   % tag_i = adopt %
(26)              end if
(27)    else oper_i ← my_op_i                                           % ac_op_i ∈ ℓ_hist_i %
(28) end if;
(30) GSTATE[i] ← ℓ_hist_i;                                   % globally update my current view %
(31) if (res ≠ ⊥) then return res to the upper layer end if
end repeat.
```

Figure 6: Obstruction-free $(1, 1)$-Universal Construction Based on Read/write Registers (code for p_i)

A Simple Obstruction-free $(1, 1)$-Universal Construction Based on Registers Only Since it is known how to solve obstruction-free consensus using registers only [15], it is possible to obtain an obstruction-free $(1, 1)$-universal construction by using an obstruction-free consensus algorithm inside Herlihy's original $(1, 1)$-universal construction [17] (or inside the three constructions presented above). However, it is possible to obtain a much simpler obstruction-free construction using only adopt-commit objects. Such a simple construction is described in Figure 6, which is a straightforward adaptation of the construction described in Figure 1, for $k = 1$. To make the understanding easier, the lines numbers used in Figure 6 are the same as the ones used in the corresponding lines of the basic construction of Figure 1.

G Discussion: Elements for a Theory of (k, ℓ)-Universality

This section sketches a few notions for a theory of (k, ℓ)-universal objects.

G.1 Definitions

The following definitions are generalizations of the notions of universal objects and consensus number introduced in [17]. They boil down to these notions when $k = \ell = 1$.

Progress condition The following definitions generalize to the universal construction of k concurrent objects (each defined by a sequential specification) the definition of the classical wait-freedom, non-blocking, and obstruction-freedom progress conditions (which corresponds to the case $k = \ell = 1$). Given a collection K of k objects, $set(K)$ denotes the set of these k objects.

- A (k, ℓ)-universal construction is *ℓ-wait-free* if, in every execution and for *every* process p, there is a set $R \subseteq set(K)$ such that (a) $|R| \geq \ell$ and (b) for every object $m \in R$, process p completes an infinite number of operations on object m.
- A (k, ℓ)-universal construction is *ℓ-non-blocking* if, in every execution, there is a set $R \subseteq set(K)$ such that (a) $|R| \geq \ell$ and (b) for every object $m \in R$, *some* process completes an infinite number of operations on object m.
- A (k, ℓ)-universal construction is *ℓ-obstruction-free* if, in every execution, there is a set $R \subseteq set(K)$ such that (a) $|R| \geq \ell$ and (b) for every object $m \in R$, *any* process completes an infinite number of operations on object m if all the other processes "hold still" long enough.

The notion of (k, ℓ)-Universality An object type T is *(k, ℓ)-universal* for n processes if, for any set of k objects, each defined by a sequential specification, there is an *ℓ-wait-free* construction of these k objects from objects of type T and atomic registers. It follows from this definition that (k, ℓ)-SC objects are (k, ℓ)-universal.

The following corollary is a simple reformulation of Theorem 4.

Corollary 3 *Let $k \geq \ell \geq 1$. The (k, ℓ)-SC object is (k, ℓ)-universal in a system of n processes, for any positive integer n.*

The notion of a (k, ℓ)-Consensus Number The (k, ℓ)-*consensus number* of an object type T, denoted $CN_{k,\ell}(T)$, is the largest n for which it is possible to wait-free implement a (k, ℓ)-SC object for n processes using any number of objects of type T and atomic registers.

Theorem 5 *Let $k \geq \ell \geq 1$. An object type T is (k, ℓ)-universal in a system of n processes if and only if $CN_{k,\ell}(T) \geq n$.*

Proof Direction $\Rightarrow$. If an object type T is (k, ℓ)-universal in a system of n processes, then (by definition), it can be used to implement a (k, ℓ)-universal construction in a system of n processes. It then follows from Theorem 4 that (k, ℓ)-SC object can be ℓ-wait-free implemented in a system of n processes. Hence, $CN_{k,\ell}(T) \geq n$.

Direction $\Leftarrow$. As (Corollary 3) (k, ℓ)-SC is (k, ℓ)-universal in a system of n processes, for any positive integer n, any object that can ℓ-wait-free implement (k, ℓ)-SC must also be (k, ℓ)-universal in a system of n processes. If $CN_{k,\ell}(T) \geq n$, it follows that (k, ℓ)-SC has an ℓ-wait-free implementation from atomic registers and objects of type T. Hence, T is (k, ℓ)-universal in a system of n processes. $\square_{Theorem\ 5}$

G.2 The relative power of object types

The importance of the notion of (k, ℓ)-consensus number as a tool for exploring the relative power of different object types ids captured by the following theorems.

Theorem 6 *Let $T1$ and $T2$ be two object types such that $CN_{k,\ell}(T1) < CN_{k,\ell}(T2)$. Then, in a system of $CN_{k,\ell}(T2)$ processes:*

- *There is no wait-free implementation of objects of type $T2$ from objects of type $T1$ and atomic registers.*

- *There is a wait-free implementation of objects of type $T1$ from objects of type $T2$ and atomic registers.*

Proof From the definition of a (k,ℓ)-consensus number it follows that it is not possible to wait-free implement a (k,ℓ)-SC object using objects of type $T1$ and registers in a system with $CN_{k,\ell}(T2)$ processes. However, it is possible to wait-free implement a (k,ℓ)-SC object using objects of type $T2$ and registers in a system with $CN_{k,\ell}(T2)$ processes. Thus, it is not possible to wait-free implement object of type $T2$ from objects of type $T1$ and registers.

By Theorem 5, if, in a system with at most n processes, $CN_{k,\ell}(T) = n$ for an object type T, any set of k objects (each defined from a sequential specification) has an ℓ- wait-free linearizable implementation using atomic registers and objects of type T. This implies that there is a wait-free implementation of an object of type $T1$ from objects of type $T2$ and atomic registers. $\Box_{Theorem\ 6}$

The previous theorem addresses the relative power of object types with different (k,ℓ)-consensus numbers. The next theorem is about object types with the same (k,ℓ)-consensus number.

Theorem 7 *Let $T1$ and $T2$ be two object types such that $CN_{k,\ell}(T1) = CN_{k,\ell}(T2) = n$. Then, in a system of at most n processes, using atomic registers, an object of type $T2$ can be wait-free implemented from objects of type $T1$ and vice-versa.*

Proof By Theorem 5, if, in a system with at most n processes, $CN_{k,\ell}(T) = n$ for an object type T, then any set of k objects (each defined sequential specification) has an ℓ-wait-free linearizable implementation using atomic registers and objects of type T. This implies that there is a wait-free implementation of an object of type $T1$ from objects of type $T2$ and atomic registers, and vice-versa. $\Box_{Theorem\ 7}$

G.3 Hierarchies

For every $k \geq \ell \geq 1$, the (k,ℓ)-*consensus hierarchy* (also called (k,ℓ)-*wait-free hierarchy*) is an infinite hierarchy of object types such that the the objects at level x of the hierarchy are exactly those types whose (k,ℓ)-consensus number is x.

In the (k,ℓ)-consensus hierarchy (1) no object type at one level together with registers can implement any object type at a higher level, and (2) each object type at one level together with registers can implement any object type at a lower level. Classifying object types by their (k,ℓ)-consensus numbers is a powerful technique for understanding the relative power of concurrent objects.