Distributed Universality

Michel Raynal[1,2], Julien Stainer[2], and Gadi Taubenfeld[3]

[1] Institut Universitaire de France
[2] IRISA, Université de Rennes 35042 Rennes Cedex, France
[3] The Interdisciplinary Center, PO Box 167, Herzliya 46150, Israel
{raynal,julien.stainer}@irisa.fr, tgadi@idc.ac.il

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)-$universal$.

Keywords: Asynchronous read/write system, universal construction, consensus, distributed computability, k-simultaneous consensus, wait-freedom, non-blocking, obstruction-freedom, contention-awareness, crash failures, state machine replication.

M.K. Aguilera et al. (Eds.): OPODIS 2014, LNCS 8878, pp. 469–484, 2014.

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 most extensively studied, and strongest 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 [14]. 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 [17]. 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 [15]. Obstruction-freedom does not guarantee progress under contention.

It has been shown in [10,14,19] 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 [14].

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 implementations of arbitrary objects into wait-free concurrent implementations of the same objects. It is shown in [14] 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 [14], several other wait-free universal constructions were proposed, which address additional properties. As an example, a universal construction is presented in [8], 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 [2,9].

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., [14,19]) of building a wait-free consensus object in $\mathcal{ARW}_{n,n-1}[\emptyset]$ relies on the notion of valence introduced in [10], 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 [5,16,26].

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. 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 consequently computationally strictly stronger than the basic model $\mathcal{ARW}_{n,n-1}[\emptyset]$.

Universal construction for k objects. An interesting question introduced in [12] 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" [12]. In their paper, Gafni and Guerraoui focus on the replication of k state machines. They present a k-universal construction, based on the replication –at every process– of each of the k state machines.

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 [17]. This construction is based on a generalization of the k-simultaneous consensus object (see below). The noteworthy features of this construction are the following.

- On the object side, 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.
- On the process side, 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.

[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 [6,24]).

- 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 [3,21,22,27].
- The construction is *generous*[3] 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.

This new universal construction is consequently called a *wait-free contention-aware obstruction-free-generous* (k, ℓ)*-universal construction*. Differently, the universal construction presented in [12] 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 [25]), we do not see how to modify the construction from [12] to overcome drawback (c).

When considering the special case $k = \ell = 1$, Herlihy's construction is wait-free $(1, 1)$-universal [14], 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

[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.

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.

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, (1) all proofs, (2) 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, and (3) definitions and notions which can be used to establish a (k, ℓ)-universality theory, are presented in [25].

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

2.1 Basic Read/Write Model and Enriched Model

The basic model presented in the introduction is the wait-free asynchronous read/write model denoted $\mathcal{ARW}_{n,n-1}[\emptyset]$ (see also [4,20,23]). 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 [18], 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 [11]. 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 $(commit, v)$, the only pairs that can be returned are $(commit, v)$ and $(adopt, v)$.
- 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., [11,23]). Hence, they provide processes only 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 [17].
- 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 [17] 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 [15] guarantees that any correct process can complete its operations if it executes in isolation for a long enough period (long enough period during which the other processes stop progressing).

3 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 SWMR (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 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 that, for each object m, are used to save the pairs $(tag, operation)$ returned by the invocation of $AC[r][m]$ of current 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

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.

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.

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

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 [12].)

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) 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

```
for each m ∈ {1,...,k} do
    my_op_i[m] ← my_list_i[m].next(); oper_i[m] ← my_op_i[m] 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 j ∈ {1,...,n} do g_state_i[j] ← GSTATE[j] end for;
     % the read of each GSTATE[j] is atomic %
(6)  for each m ∈ {1,...,k} do
(7)    ℓ_hist_i[m] ← longest history of g_state_i[1..n][m] containing ℓ_hist_i[m];
(8)    if (my_op_i[m] ∈ ℓ_hist_i[m])                    % my operation was completed %
(9)          then output_i[m] ← compute_output(my_op_i[m], ℓ_hist_i[m]);
(10)              return {(m, my_op_i[m], output_i[m])} to the upper layer;
(11)              my_op_i[m] ← my_list[m].next()
(12)   end if
(13) end for;

(14) res ← ∅;
(15) for each m ∈ {1,...,k} do
(16)    if (ac_op_i[m] ∉ ℓ_hist_i[m])                   % operation was not completed %
(17)      then if (tag_i[m] = commit)                      % complete the operation %
(18)             then ℓ_hist_i[m] ← ℓ_hist_i[m].append(ac_op_i[m]);
(19)                  if (ac_op_i[m] = my_op_i[m])       % my operation was completed %
(20)                    then output_i[m] ← compute_output(ac_op_i[m], ℓ_hist_i[m]);
(21)                         res ← res ∪ {(m, my_op_i[m], output_i[m])};
(22)                         my_op_i[m] ← my_list[m].next()
(23)                  end if;
(24)                  oper_i[m] ← my_op_i[m]
(25)             else oper_i[m] ← ac_op_i[m]                  % tag_i[m] = adopt %
(26)           end if
(27)      else oper_i[m] ← my_op_i[m]                  % ac_op_i[m] ∈ ℓ_hist_i[m] %
(28)    end if
(29) end for;

(30) GSTATE[i] ← ℓ_hist_i[1..k];                    % globally update my current view %
(31) if (res ≠ ∅) then return res to the upper layer end if
end repeat.
```

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

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.

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 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 [27].)

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

```
for each m ∈ {1,...,k} do
      my_op_i[m] ← my_list_i[m].next(); oper_i[m] ← my_op_i[m] end for.

repeat forever
(1)   r_i ← r_i + 1;
(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;
(N2)  if (∃ m ∈ {1,...,k} : tag_i[m] = adopt) then
(2M)      (ksc_obj, ksc_op) ← kSC[r_i].propose(ac_op_i[1..k]);
(3M)      (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
(N3)  end if;
lines 5- 31 of the construction of Figure 1
end repeat.
```

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

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).

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.*

It is possible to still reduce the number of uses of underlying k-SC synchronization objects. by replacing the lines N1-N3 in Figure 2 as described in Figure 3. There is one modified line (N2M) and three new lines (NN1, NN2, and NN3). 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).

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

```
(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.
```

Fig. 3. Efficient Contention-aware Non-Blocking $(k, 1)$-Universal Construction (code for p_i)

concerned, it is important to notice that 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 $my\_list_i.\mathsf{next}()$ (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 $\mathsf{next}()$.

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 ← |ℓ_hist_i[m]| mod n + 1; next_prop_m ← LAST_OP[j, m];
(L2) if next_prop_m ∉ ({⊥} ∪ ℓ_hist_i[m])
(L3)    then oper_i[m] ← next_prop_m
(L4)    else oper_i[m] ← my_op_i[m]
(L5) end if.
```

This helping mechanism is close to the one proposed in [14]. 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 $my\_list_j[m].\mathsf{next}()$. 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 *define a wait-free contention-aware linearizable* $(k, 1)$*-universal construction.*

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 $(x_1, v_1), ..., (x_\ell, v_\ell)$ (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 on top of $\mathcal{ARW}_{n,n-1}[(k, \ell)\text{-}SC]$ as follows. 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 whose first components are all distinct. It is 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., $AC[2r_i][ksc\_obj_x].\mathsf{propose}(ksc\_op_x), 1 \leq x \leq \ell$) can be executed in parallel, which means in any order. Let us notice that if several processes invokes $AC[2r_i][ksc\_obj_x].\mathsf{propose}()$ on the same object ksc_obj_x, they invoke it with the same operation ksc_op_x.
- Line 4M: $AC[2r_i][m].\mathsf{propose}(oper_i[m])$ 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 $AC[2r_i][ksc\_obj_x].\mathsf{propose}()$ 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.

[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.

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

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. The constructions presented in the paper can be seen as innovative generalizations of the universality notions introduced in [12,14]. More specifically, we have presented the following suite of constructions:

- 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, a simple obstruction-free $(1,1)$-universal construction based on atomic registers only, and elements for a theory of (k,ℓ)-universality can be found in [25].

References

1. Afek, Y., Gafni, E., Rajsbaum, S., Raynal, M., Travers, C.: The k-simultaneous consensus problem. Distributed Computing 22(3), 185–195 (2010)
2. Anderson, J.H., 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., Kutnetsov, P.: The complexity of obstruction-free implementations. Journal of the ACM 56(4), Article 24, 33 (2009)
4. Attiya, H., Welch, J.L.: Distributed computing: Fundamentals, simulations and advanced topics, 2nd edn., p. 414. Wiley Interscience (2004) ISBN 0-471-45324-2
5. Borowsky, E., Gafni, E., Generalized, F.L.P.: impossibility results for t-resilient asynchronous computations. In: Proc. 25th ACM Symposium on Theory of Computing (STOC 1993), pp. 91–100. ACM Press (1993)

6. Bouzid, Z., Travers, C.: Simultaneous consensus is harder than set agreement in message-passing. In: Proc. 33rd Int'l IEEE Conference on Distributed Computing Systems (ICDCS 2013), pp. 611–620. IEEE Press (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. Ellen, F., Fatourou, P., Kosmas, E., Milani, A., Travers, C.: Universal construction that ensure disjoint-access parallelism and wait-freedom. In: Proc. 31th ACM Symposium on Principles of Distributed Computing (PODC), pp. 115–124. ACM Press (2012)
9. Fatourou, P., Kallimanis, N.D.: A highly-efficient wait-free universal construction. In: Proc. 23th ACM Symposium on Parallel Algorithms and Architectures (SPAA), pp. 325–334. ACM Press (2012)
10. Fischer, M.J., Lynch, N.A., Paterson, M.S.: Impossibility of distributed consensus with one faulty process. Journal of the ACM 32(2), 374–382 (1985)
11. Gafni, E.: Round-by-round fault detectors: unifying synchrony and asynchrony. In: Proc. 17th ACM Symp. on Principles of Distr. Computing (PODC), pp. 143–152. ACM Press (1998)
12. Gafni, E., Guerraoui, R.: Generalized universality. In: Katoen, J.-P., König, B. (eds.) CONCUR 2011. LNCS, vol. 6901, pp. 17–27. Springer, Heidelberg (2011)
13. Guerraoui, R., Kapalka, M., Kouznetsov, P.: The weakest failure detectors to boost obstruction-freedom. Distributed Computing 20(6), 415–433 (2008)
14. Herlihy, M.P.: Wait-free synchronization. ACM Transactions on Programming Languages and Systems 13(1), 124–149 (1991)
15. Herlihy, M.P., Luchangco, V., Moir, M.: Obstruction-free synchronization: double-ended queues as an example. In: Proc. 23th Int'l IEEE Conference on Distributed Computing Systems (ICDCS 2003), pp. 522–529. IEEE Press (2003)
16. Herlihy, M.P., Shavit, N.: The topological structure of asynchronous computability. Journal of the ACM 46(6), 858–923 (1999)
17. Herlihy, M.P., Wing, J.M.: Linearizability: A correctness condition for concurrent objects. ACM Transactions on Programming Languages and Systems 12(3), 463–492 (1990)
18. Lamport, L.: On inter-process communications, Part I: Basic formalism. Distributed Computing 1(2), 77–85 (1986)
19. Loui, M., Abu-Amara, H.: Memory requirements for agreement among unreliable asynchronous processes. Advances in Computing Research 4, 163–183 (1987)
20. Lynch, N.A.: Distributed algorithms, vol. 872. Morgan Kaufmann (1996)
21. Luchangco, V., Moir, M., Shavit, N.N.: On the Uncontended complexity of consensus. In: Fich, F.E. (ed.) DISC 2003. LNCS, vol. 2848, pp. 45–59. Springer, Heidelberg (2003)
22. Merritt, M., Taubenfeld, G.: Resilient consensus for infinitely many processes. In: Fich, F.E. (ed.) DISC 2003. LNCS, vol. 2848, pp. 1–15. Springer, Heidelberg (2003)
23. Raynal, M.: Concurrent programming: Algorithms, principles, and foundations, 515 p. Springer (2013) ISBN 978-3-642-32026-2
24. Raynal, M., Stainer, J.: Simultaneous consensus vs set agreement: A message-passing-sensitive hierarchy of agreement problems. In: Moscibroda, T., Rescigno, A.A. (eds.) SIROCCO 2013. LNCS, vol. 8179, pp. 298–309. Springer, Heidelberg (2013)
25. Raynal, M., Stainer, J., Taubenfeld, G.: Distributed universality. Tech Report, pages, IRISA, Université de Rennes, France (2014)
26. Saks, M., Zaharoglou, F.: Wait-free k-set agreement is impossible: the topology of public knowledge. SIAM Journal on Computing 29(5), 1449–1483 (2000)
27. Taubenfeld, G.: Contention-sensitive data structures and algorithms. In: Keidar, I. (ed.) DISC 2009. LNCS, vol. 5805, pp. 157–171. Springer, Heidelberg (2009)