Distributed Universal Constructions: a Guided Tour

Michel Raynal

To cite this version:

Distributed Universal Constructions: a Guided Tour

Michel Raynal

Institut Universitaire de France
IRISA, Université de Rennes, 35042 Rennes, France
Department of Computing, Hong Kong Polytechnic University

Tech Report 2040, November 2016
IRISA, University of Rennes (France)

November 21, 2016

Abstract

The notion of a universal construction is central in computing science: the wheel has not to be reinvented for each new problem. In the context of n-process asynchronous distributed systems, a universal construction is an algorithm that is able to build any object defined by a sequential specification despite the occurrence of up to $(n-1)$ process crash failures. The aim of this paper is to present a guided tour of such universal constructions. Its spirit is not to be a catalog of the numerous constructions proposed so far, but a (as simple as possible) presentation of the basic concepts and mechanisms that constitute the basis these constructions rest on.

Keywords: Abortable object, Agreement problem, Asynchronous read/write system, Atomic operations, Computability, Concurrent object, Consensus, Crash failure, Disjoint-access parallelism, Help mechanism, LL/SC instruction, Memory location, Non-blocking, Obstruction-freedom, Progress condition, Sequential specification, k-Set agreement, k-Simultaneous consensus, Speculative execution, Universal construction, Wait-freedom.

1 Introduction

A (very) short historical perspective Looking for (some) universality seems inherent to humankind. Any language, any writing system, can be seen as an attempt to universality [42]. In the science domain, one of the very first witness of research of universality found in the past seems to be the Plimpton 322 tablet (Figure 1), which describes the fifteen first Pythagorean triplets ($a^2 + b^2 = c^2$).This is only a list, not yet an algorithm with its proof.

Figure 1: Plimpton 322 tablet

The geometric constructions with a compass and a straightedge designed by the Ancient Greeks are among the first algorithms with their correctness proofs (see also [50]). Proofs of impossible constructions in the "compass + straightedge" computing model took more time (e.g., the impossibility of squaring the circle, i.e., build, with straightedge and compass only, a square whose area is equal to the area of a given circle)[1]. More recently, the Turing machine provides us with an abstract computing device, which is considered as the most general sequential computing model, thereby fixing the limits of what can be computed by a sequential machine [61][2]. It is consequently claimed to be *universal*. The *halting* problem is the most famous of the problems that are impossible to solve in this "most general" sequential computing model.

In distributed computing the situation is different. As written in [36]: "*In sequential systems, computability is understood through the Church-Turing Thesis: anything that can be computed, can be computed by a Turing Machine. In distributed systems, where computations require coordination among multiple participants, computability questions have a different flavor. Here, too, there are many problems which are not computable, but these limits to computability reflect the difficulty of making decisions in the face of ambiguity, and have little to do with the inherent computational power of individual participants.*"

In distributed computing the main issues posed by universality and computability appear when one has to implement distributed state machines (distributed services encapsulated in concurrent objects) in the presence of adversaries due to the environment in which the computation evolves (such as asynchrony and process failures) [25, 32, 43, 46].

[1] This impossibility follows from the fact that π is a transcendent number (F. von Lindemann 1882), and a theorem by P. L. Wantzel, who established, in 1937, necessary and sufficient conditions for a number to be constructible in the "compass + straightedge" computing model [62].

[2] This means that any sequential computing model proposed so far has the same computability power as a Turing machine (e.g., Church's Lambda calculus, or Post systems [51]), or is weaker than a Turing machine (e.g., finite state automata).

Concurrent objects and asynchronous crash-prone read/write systems A concurrent object is an object that can be accessed (possibly simultaneously) by several processes. From both practical and theoretical point of views, a fundamental problem of concurrent programming consists in implementing high level concurrent objects, where "high level" means that the object provides the processes with an abstraction level higher than the atomic hardware-provided instructions. While this is well-known and well-mastered since a long time in the context of failure-free systems [13], it is far from being trivial in failure-prone systems (e.g., see textbooks such as [52, 58]), where it is still an important research domain.

This paper considers systems made up of n sequential asynchronous processes which, at the hardware level, communicate through memory locations (memory words also called registers) which can be accessed by atomic operations (instructions), including the basic read and write operations. Moreover, it is assumed that, in any run, up to $(n-1)$ processes may crash (unexpected halting). When restricted to the basic read and write instructions, this computation model is known under the name *wait-free read/write* model (denoted here $\mathcal{CARW}_n[\emptyset]$, where $\mathcal{CARW}$ stands for Crash Asynchronous Read/write).

On progress conditions Deadlock-freedom and starvation-freedom are well-known progress conditions in failure-free asynchronous systems. As their implementation is based on lock mechanisms, they are not suited to asynchronous crash-prone systems. This is due to the fact that it is impossible to distinguish a crashed process from a slow process, and consequently a process that acquires a lock and crashes before releasing it can entail the blocking of the entire system.

Hence, new progress conditions for concurrent objects suited to crash-prone asynchronous systems have been proposed. Given an object, we have the following.

- The strongest progress condition is *wait-freedom* (WF) [32]. It states that, any operation (on the object that is built) issued by a process that does not crash terminates. This means that it terminates whatever the behavior of the other processes. This can be seen as the equivalent of the starvation-freedom progress condition encountered in failure-free systems.
- The *non-blocking* progress condition (NB) states that there is at least one process that can always progress (all its object operations terminate) [38]. This progress condition is also called *lock-freedom*. It can be seen as the equivalent of deadlock-freedom in failure-free systems. Non-blocking has been generalized in [14], under the name k*-lock-freedom* (k-NB), which states that at least k processes can always make progress.
- The *obstruction-freedom* progress condition (OB) states that a process that does not crash will be able to terminate its operation if all the other processes hold still long enough [34]. This is the weakest progress condition. It has been generalized in [59], under the name k*-obstruction-freedom* (k-OB), which states that, if a set of at most k processes run alone for a sufficiently long period of time, they will terminate their operations.

While *wait-freedom* and *non-blocking* are independent of the concurrency and failure pattern, *obstruction-freedom* is dependent from it. Asymmetric progress conditions have been introduced in [41]. The computational structure of progress conditions is investigated in [60].

Universal construction The notion of a universal construction was introduced by M. Herlihy in [32]. It considers objects (a) which are defined from sequential specifications and (b) whose operations are total, i.e., any object operation returns a result (as an example, a $\mathsf{push}()$ operation on an empty stack returns the default value $\perp$).

Let PC be a progress condition. A *PC-compliant universal construction* is an algorithm that, given the sequential specification of an object O (or a sequential implementation of it), provides a concurrent implementation of O satisfying the progress condition PC in the the presence of up $(n-1)$ process crashes (Figure 2).

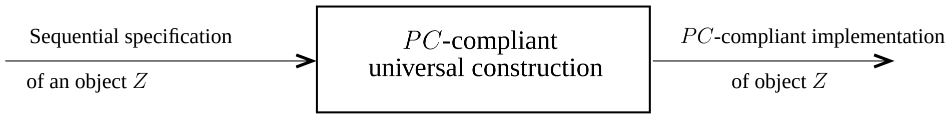

Figure 2: PC-compliant universal construction

It has been shown in [25, 32, 47] that the design of a universal construction with respect to the wait-freedom progress condition is impossible in $\mathcal{CARW}_n[\emptyset]$. This means that the basic system model $\mathcal{CARW}_n[\emptyset]$ has to be enriched with hardware-provided atomic instructions or additional computing objects whose computational power is stronger than atomic read/write registers (in the following, we consider terms "register" and "memory location" as synonyms; we sometimes also say "atomic read/write object" by a slight abuse of language).

Content of the paper This paper aims at being a guided tour to distributed universal constructions. Its goal is not to be a presentation including as many universal constructions as possible, but to focus on the central features universal constructions rest on, and illustrate them with existing algorithms. To this end, after having introduced basic definitions (Section 2), the paper proceeds as follows.

- Section 3 presents first a simple and elegant universal construction suited to the system model $\mathcal{CARW}_n$[LL/SC] ($\mathcal{CARW}_n[\emptyset]$ enriched with the hardware-provided instructions LL and SC, which are defined in the section). This allows for an easy introduction of the notion of a *speculative computation* and the notion of a *help mechanism* (introduced in [32] and recently formalized in [17]). This section presents also extensions devoted to *large* objects.
- Section 4 is made up of two subsections. the first is on the efficiency of universal constructions. Considering the algorithms that realize them, it addresses the notion of *disjoint-access parallelism*.

  The second subsection is on the object side. It considers the case of universal constructions for deterministic *abortable* objects [15, 31, 52, 53]. Such an object is a classical object defined by a sequential specification which allows an operation to return a default value $\perp$ in the presence of contention (in this case the operation has no effect on the object). Hence, in a concurrency-free execution, an abortable object behaves as its non-abortable counterpart. The notion of k-abortable object has been recently introduced in [8], where is also presented an associated universal construction. A k-abortable object is such that an operation is allowed to return $\perp$ only if it is concurrent with operations from at most k different processes, and none these operations return $\perp$.
- All the previous universal constructions consider that the underlying crash-prone system is enriched with hardware-provided atomic instructions such as LL/SC or Compare&Swap, which work on memory locations [22]. Hence, the question: Which are the instructions that allow to build a universal construction? As an example, can a universal construction be designed for the system model $\mathcal{CARW}_n$[Test&Set] ($\mathcal{CARW}_n[\emptyset]$ enriched with the hardware-provided atomic instruction Test&Set). This issue was solved by M. Herlihy in [32], who introduced the celebrated *consensus hierarchy*. This is addressed in the first part of Section 5. Hence, the consensus object is at the core of universal constructions.

  Then, the section shows another important advantage of using consensus objects instead of primitives hardware-provided instructions to design universal constructions. While instructions are uniform (any instruction can access any memory location [22]), an object is a typed abstraction that has the property that an operation on type T cannot be applied to an object of type T'.

Moreover, an object can be weakened or generalized according to the needs of the user. As an example, the consensus object can be weakened to the k-set agreement (k-SA) object [19] or to the k-simultaneous consensus (k-SC) object [3] (k-SA and k-SC objects are defined in the section).

The section presents then the notion of a k-universal construction due to E. Gafni and R. Guerraoui [27]. Such a construction considers k objects (instead of only one) and ensures that at least one of these objects progresses forever. This construction relies on k-SC objects instead of consensus objects.

Finally, the section considers the case where we want that, not at least one but at least ℓ objects progress forever, where ℓ is any predefined constant in $[1..k]$. As shown in [55], objects denoted (k, ℓ)-SC ((k, ℓ)-simultaneous consensus objects defined in the section), which are strictly stronger than k-SC objects (when $\ell > 1$), and weaker than consensus objects (when $\ell < k$), are necessary an sufficient to build a universal construction for k objects, where at least ℓ objects progress forever. It is important to notice that these generalizations of universal constructions could not have been obtained from hardware-provided instructions. This will conclude the guided tour.

Finally, after a short Section 6 comparing universal constructions and software transactional memory (STM) systems, Section 7 concludes the paper.

2 The Basic Asynchronous Read/Write Model $\mathcal{CARW}_n[\emptyset]$

Crash-prone asynchronous processes The basic computing model denoted $\mathcal{CARW}_n[\emptyset]$ was sketched in the introduction. It is composed of a set of n sequential processes denoted p_1, ..., p_n. Each process is asynchronous which means that it proceeds at its own speed, which can be arbitrary and remains always unknown to the other processes.

A process may halt prematurely (crash failure), but executes correctly its local algorithm until it possibly crashes. Up to $(n-1)$ processes may crash in a run. Due to the atomicity of the hardware-provided operations, if a process crashes while executing such an operation, this operation appears as entirely executed or not at all. A process that crashes in a run is said to be *faulty* in this run. Otherwise, it is *correct* or *non-faulty*. Hence, a faulty process is a process whose speed, after some time, remains forever equal to 0.

On atomicity The processes communicate by accessing atomic read/write registers (memory locations). Atomicity means that the read and write primitive operations on a register appear as if they have been executed one after the other. Moreover, the corresponding sequence of operations S is such that (a) if the operation op_1 terminated before the operation op_2 started, op_1 appears before op_2 in S, and (b) a read operation on a register R returns the value written by the closest preceding write operation on R (or its initial value if there is no preceding write) [44]. Atomicity is also called *linearizability* when considering any object defined by a sequential specification [38].

Notation Variables local to a process p_i are denoted with lowercase letters, sometimes indexed with i. Memory location and objects shared by the processes are denoted with capital letters.

3 A Simple LL/SC-Based WF-Compliant Universal Construction

3.1 Extending $\mathcal{CARW}_n[\emptyset]$ with LL/SC

Model $\mathcal{CARW}_n$[LL/SC] These hardware-provided atomic instructions can be applied to any memory location. The wait-free read/write model $\mathcal{CARW}_n[\emptyset]$ enriched with them is denoted $\mathcal{CARW}_n$[LL/SC].

LL/SC is made up of three instructions: LL stands for Linked Load; SC stands for store conditional; VL stands for Validate.

Let X be a memory location. $X.\mathsf{LL}()$ returns the current value of X. Let p_i be a process that invokes $X.\mathsf{SC}(v)$. This invocation assigns v to X if X has not been assigned a value by another process since the previous invocation of $X.\mathsf{LL}()$ issued by p_i. In this case, $X.\mathsf{SC}(v)$ returns `true` and we say that the invocation is successful; otherwise it returns `false`. Finally, an invocation of $X.\mathsf{VL}()$ by process p_i returns `true` if no other process has issued a successful $X.\mathsf{SC}()$ since the last invocation of $X.\mathsf{LL}()$ issued by p_i.

These instructions are used to bracket a *speculative computation*. A process first reads X with $X.\mathsf{LL}()$ and stores its value in a local variable x_i. Then p_i does a local computation which depends on both x_i and its local state. The aim of this local computation is to define a new value v for X. Finally, p_i tries to commit its local computation by writing v into X, which is done by invoking $X.\mathsf{SC}(v)$. If this invocation is successful, the write is committed; otherwise the write fails. A similar behavior can be obtained by the Compare&Swap() instruction. The main advantage of LL/SC, with respect to Compare&Swap(), is that it does suffer the ABA problem (which requires sequence numbers to be solved) [52, 58]. Algorithms based on LL/SC can be found in many publications (e.g., [23, 33, 39, 52, 58, 59] to cite a few).

3.2 A simple universal construction in $\mathcal{CARW}_n[$**LL/SC**$]$

This section presents a simplified version (denoted sFK) of a universal construction due to P. Fatourou and N. Kallimanis [24]. The main difference is that the presented construction uses sequence numbers which increase forever, while [24] uses sequence numbers modulo 2). This additional memory cost makes it much easier to present and prove correct.

Collect object This construction uses a collect object. Such an object can easily be built in $\mathcal{CARW}_n[\emptyset]$. It consists of an array $COL[1..n]$, with one entry per process, and provides them with two operations denoted $\mathsf{update}()$ and $\mathsf{collect}()$. The invocation of $COL.\mathsf{update}(v)$ by a process p_i assigns v to $COL[i]$. The invocation of $COL.\mathsf{collect}()$ is an asynchronous scan of the array which returns, for each entry j, the value it has read from $COL[j]$. A formal definition of such an object can be found in [52].

Due to the asynchronous scan, a collect object is not atomic (hence a collect object is computationally weaker than a snapshot object [1]). An atomic version of a collect object is described in [24]. Its implementation (a) assumes that the n components of the collect object are stored in a single memory location, and (b) is based on the hardware-provided instruction $\mathsf{add}()$ ($Y.\mathsf{add}(\mathsf{v})$ atomically adds v to Y).

Global and local variables Let O be the object that is built.

- $STATE$ is a memory location made up of three fields:
  - $STATE.value$ contains the current value of O. It is initialized to the initial value of O.
  - $STATE.sn[1..n]$ is an array of sequence numbers initialized to $[0, \cdots, 0]$; $STATE.sn[i]$ is the sequence number of the last invocation of an operation on O issued by p_i.
  - $STATE.res[1..n]$ is an array of result values initialized to $[\bot, \cdots, \bot]$; $STATE.res[i]$ contains the result of the last operation issued by p_i that has been applied to O.
- $BOARD$ is a collect object. Each of its entry i contains a pair $\langle BOARD[i].op, BOARD[i].sn\rangle$ initialized to $\langle \bot, 0\rangle$; $BOARD[i].op$ contains the last operation on O issued by p_i, and $BOARD[i].sn$ contains its sequence number.
- Each process p_i manages a sequence number generator sn_i initialized to 1.

The object O is assumed to be defined by a transition function $\delta()$. Let s be the current state of O and $\mathsf{op}(in)$ be the invocation of the operation $\mathsf{op}()$ on O, with input parameter in; $\delta(s, \mathsf{op}(in))$ outputs a pair $\langle s', r\rangle$ such that s' is the state of O after the execution of $\mathsf{op}(in)$ on s, and r is the result of $\mathsf{op}(in)$.

Construction sFK: speculative computation and helping The construction sFK is described in Figure 3. When a process p_i invokes an operation $\mathsf{op}(in)$ on O, it first publishes the pair $\langle \mathsf{op}(in), sn_i\rangle$ in the collect object $BOARD$ (line 1). Then, it invokes the internal procedure $\mathsf{apply}()$ at the end of which it will locally return the result produced by $\mathsf{op}(in)$ (line 2).

```
when $p_i$ invokes $\mathsf{op}(in)$ do
(1)   $BOARD.\mathsf{update}(\langle \mathsf{op}(in), sn_i\rangle)$; $sn_i \leftarrow sn_i + 1$;
(2)   $\mathsf{apply}()$; let $r$ = $STATE.res[i]$; $\mathsf{return}(r)$.

internal procedure $\mathsf{apply}()$ is
(3)   repeat twice
(4)       $ls \leftarrow STATE.\mathsf{LL}()$;
(5)       $pairs \leftarrow BOARD.\mathsf{collect}()$;
(6)       for $\ell \in \{1, 2, \cdots, n\}$ do
(7)           if $(pairs[\ell].sn = ls.sn[\ell] + 1$ then
(8)               $\langle new\_state, r\rangle \leftarrow \delta(ls.value, pairs[\ell].op)$;
(9)               $ls.res[\ell] \leftarrow r$; $ls.sn[\ell] \leftarrow pairs[\ell].sn$
(10)          end if
(11)      end for
(12)      $STATE.\mathsf{SC}(ls)$
(13)  end repeat twice.
```

Figure 3: WF-compliant universal construction sFK (system model $\mathcal{CARW}_n$[LL/SC])

The core of the construction is the procedure $\mathsf{apply}()$, in which a process p_i executes twice the lines 4-12 (we will see later why this has to be done twice). Process p_i first reads the current local state of the object (line 4), and starts a first speculative execution (which will end at line 12). In this speculative execution, p_i first reads the content of the collect object $BOARD$ from which it obtains for each process p_ℓ a pair $\langle$last operation invoked by p_ℓ, associated sequence number$\rangle$. Let us remind that as $BOARD.\mathsf{collect}()$ is not atomic, and p_i is asynchronous, the pairs that are returned are not necessarily associated with a consistent global state the computation passed through.

Then, p_i considers each pair in $pairs$ in the "for" loop of lines 6-11. In this loop, p_i strives to help all the processes that have a pending operation on O. From its point of view (i.e., with the information it has obtained from its previous reads of $STATE$ and $BOARD$), those are all the processes p_ℓ such that $pairs[\ell].sn = ls.sn[\ell] + 1$ (line 7). If this local predicate is true, p_i locally simulates (speculative computation) the last operation issued by p_ℓ not yet applied to the object (line 6), and locally saves the result of the operation and its sequence number (line 9). Finally, p_i tries to commit its speculative computation by invoking $STATE.\mathsf{SC}()$ (line 12). Let us observe that, if this invocation is successful, we can conclude that no process modified $STATE$ while p_i was doing its speculative computation. Hence, the local variable ls of p_i is up to date, and, from an external observer point of view, everything appears as if the computation starting at line 4 and ending at line 12 was executed atomically. If the invocation of $STATE.\mathsf{SC}()$ is not successful, the speculative execution is not committed.

Construction sFK: why "repeat twice"? Let us first observe that, due to sequence numbers, once registered in the collect object $BOARD$, an operation cannot be executed once more than once. Moreover, if the process p_i that invokes an operation does not crash, it terminates its operation $\mathsf{op}(in)$. This follows from the fact that the lines 7-10 are executed a bounded number of times ($2n$). But is the result provided for $\mathsf{op}(in)$ correct?

To answer this question, let us consider the execution described in Figure 4. When process p_j (bottom of the figure) executes $STATE.\mathsf{LL}()$ followed by $BOARD.\mathsf{collect}()$ (lines 4-5), p_i (top of the figure) has not yet registered by executing $BOARD.\mathsf{update}()$ (line 1). Hence $pairs_j$ does not contain $\langle \mathsf{op}(in), sn\rangle$. Let us assume that the execution of $STATE.\mathsf{SC}(ls_j)$ by p_j is successful. If p_i executes only once the repeat loop, its execution of $STATE.\mathsf{SC}()$ is not successful, and p_i returns despite the fact that p_j has not helped it by executing $\mathsf{op}(in)$. Hence, the statement $\mathsf{return}(r)$ executed by p_i at line 2 returns the result of its previous operation invocation.

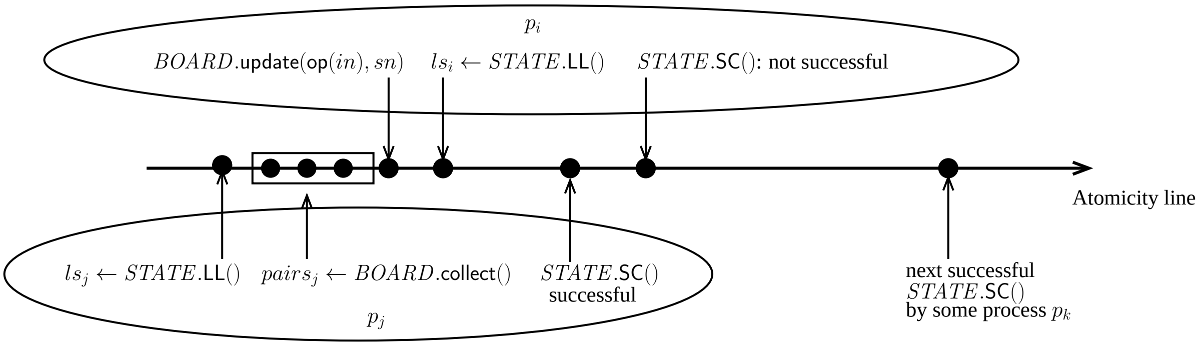
p_i
$BOARD.\mathsf{update}(\mathsf{op}(in), sn)$
$ls_i \leftarrow STATE.\mathsf{LL}()$
$STATE.\mathsf{SC}()$: not successful
Atomicity line
$ls_j \leftarrow STATE.\mathsf{LL}()$
$pairs_j \leftarrow BOARD.\mathsf{collect}()$
$STATE.\mathsf{SC}()$ successful
p_j
next successful $STATE.\mathsf{SC}()$ by some process p_k

Figure 4: Why to repeat twice lines 4-12 (big dot = atomic step)

Assuming now that p_i executes twice the repeat loop, let us consider the first successful invocation of $STATE.\mathsf{SC}()$ that occurs after the previous successful invocation by p_j. This invocation is issued by some process p_k (which can be p_i, p_j or any other process). According to the algorithm of Figure 3, it follows that p_k has previously invoked $STATE.\mathsf{LL}()$. Moreover, this invocation occurs necessarily after the successful invocation of $STATE.\mathsf{SC}()$ by p_j (otherwise the invocation of $STATE.\mathsf{SC}()$ by p_k could not be successful). Consequently, the invocation of $BOARD.\mathsf{collect}()$ by p_k is such that $\langle \mathsf{op}(in), sn\rangle \in pairs_k$. It follows that p_k found $pairs_k[i].sn = ls_k.sn[i] + 1$, and simulated the execution of $\mathsf{op}(in)$ on behalf of p_i and wrote the corresponding result in $ls_k.res[i]$ which was then copied in $STATE.res[i]$ by the successful execution of $STATE.\mathsf{SC}()$ by p_k.

Linearization of the operations on O Let SC[1], SC[2], ..., SC[x], etc. be the sequence of all the successful invocations of $STATE.\mathsf{SC}()$; as $STATE.\mathsf{SC}()$ is atomic, this sequence is well-defined. Starting from $SC[1]$, each SC[x] applies at least one operation on the object O. It is possible to totally order the operations applied to O by each SC[x]. Let seq[x] be the corresponding sequence. The sequence of operations applied to O is then seq[1] followed by seq[2], ..., followed by seq[x], etc.

Remark on sequence numbers Techniques such as the one described in [9, 48] (known under the name *alternating bit protocol*) can be used to obtain an implementation in which the sequence numbers are implemented modulo 2.

3.3 The case of large objects

The previous universal construction considered that the internal state of the object ($STATE$) can be copied all at once. A *large* object is an object whose internal state cannot be copied in one instruction.

Several articles have addressed this problem, e.g., [2, 6, 33]. They all propose to fragment a large object into blocks. Two main approaches have been proposed.

- One consists in using pointers linking the blocks representing the object [33]. Moreover, it requires that the programmer provides a sequential implementation of the object that performs as little copying as possible. The pointers are then accessed with LL instructions which allow a process to obtain a logical copy of the object (which means that only the needed part of the object is copied in its local memory). A process executes then locally a speculative computation, as defined by the operation it want to apply to the object. Finally it uses SC instructions on the appropriate pointers to try to commit the new value of the object.

- The other approach consists in representing the object as a long array fragmented into blocks [6]. This paper presents two object constructions based on this approach, which are universal with respect to non-blocking and wait-freedom, respectively. It also presents algorithms implementing atomic LLL/LSC operations (where "L" stands for Large), which extend the LL/SC instructions to arrays of memory locations. These operations are built in the system model $\mathcal{CARW}_n[\text{LL/SC}]$.

4 Extensions

This section presents two extensions of universal constructions. The first one is their efficiency. The second one considers a weakening of concurrent objects called abortable objects.

4.1 On the implementation side: Disjoint-access parallelism

Disjoint-access parallelism A universal construction is *disjoint-access parallel* if two processes that access distinct parts of an object O do not access common base objects or common memory location which constitute the internal representation of O. As an example, let us consider a queue. If the queue contains three or more items, a process executing $\mathsf{enqueue}(v)$ and a process executing $\mathsf{dequeue}()$ must be able to progress without interfering.

Hence, the aim of a disjoint-access parallel universal construction is to provide efficient implementations. Let us observe that all the universal constructions that built a total order on the operations (such as the one described in Section 3.2 and the ones presented in [2, 23, 33]) are not disjoint-access parallel.

What can be done? Hence the question posed by F. Ellen, P. Fatourou, N. Kosmas, A. Milani, and C. Travers, in [21]: Is it possible to design a disjoint-access parallel WF-compliant universal construction? This work presents two important results.

- The first is an impossibility result. It states that it is impossible to design a universal construction that is disjoint-access parallel and ensures that all the operation invocations of the processes that do not crash always terminate. Hence, when we consider any object defined by a sequential specification, disjoint-access parallelism and wait-freedom are mutually exclusive.
- The second result is a positive one, namely the previous impossibility (which considers *any* object defined by a sequential specification) does not apply to a special class of concurrent objects. Hence, the constructions for this object class are no longer "universal" in the strict sense. This object class contains all the objects O for which, in any sequential execution, each operation accesses a bounded number of base objects used to represent O. Examples of such objects are bounded trees, or stacks and queues whose internal representations are list-based.

  In their paper, the authors describe a universal construction that ensures, for the previous objects, both the disjoint-access parallel property of the object implementation, and the wait-freedom progress condition for the processes that use it. This construction is presented in the system model $\mathcal{CARW}_n[\text{LL/SC}]$.

4.2 On the object side: Abortable objects

Abortable objects have been investigated in several articles, e.g., [4, 15, 31, 52, 53]. They found their origin in the commit/abort output of transaction-based systems [28], and the notion of "fast path" initially introduced in fast mutual exclusion algorithms [45].

Definition An abortable object is an object (defined by a sequential specification) such that

- When executed in a contention-free context, an operation takes effect, i.e., modifies the state of the object and returns a result as defined by its sequential specification,
- When executed in a contention context, an operation either takes effect and returns a result as defined by its sequential specification, or returns the default value $\perp$ (abort). If $\perp$ is returned, the operation has no effect on the state of the object.

Hence, an abortable object is such that any operation always returns (i.e., whatever the concurrency context). Its progress condition is consequently wait-freedom. Differently from an abortable object, an obstruction-free object does not guarantee operation termination in the presence of concurrency. A theory of deterministic abortable objects (including a study of their respective power) is presented in [31].

Universal constructions for abortable objects Such a very simple construction is described in Figure 5. It is a trivial simplification of the universal construction described in Figure 3 from which the helping mechanism has been suppressed. The memory location $STATE$ contains now only the state of the object.

```
when $p_i$ invokes $\mathsf{op}(in)$ do
(1)   $ls \leftarrow STATE.\mathsf{LL}()$;
(2)   $\langle new\_state, r\rangle \leftarrow \delta(ls, pairs[\ell].op)$;
(3)   $done \leftarrow STATE.\mathsf{SC}(ls)$;
(4)   if $(done)$ then $\mathsf{return}(r)$ else $\mathsf{return}(\perp)$ end if.
```

Figure 5: WF-compliant universal construction for abortable objects (system model $\mathcal{CARW}_n$[LL/SC])

When a process p_i invokes an operation $\mathsf{op}(in)$ on the object, it reads its current state to obtain a local copy (line 1). Then it produces a speculative execution of $\mathsf{op}(in)$ on this local state ls (line 2). Finally, it tries to commit its local execution by issuing $STATE.\mathsf{SC}(ls)$ (line 3). If this SC is successful, p_i returns the result it has previously computed. Otherwise, there was at least one concurrent operation, and p_i returns $\perp$ (line 4).

Let us observe that, if several processes concurrently invoke operations, each invokes $STATE.\mathsf{LL}()$, and the first of them that invokes $STATE.\mathsf{SC}()$ produces a successful SC. It follows that, in the presence of concurrency, at least one process is guaranteed to make progress in the sense that it does not return $\perp$.

An efficient *solo-fast* universal construction for deterministic abortable objects is described in [15]. Solo-fast (also called contention-aware in other articles) means that the implementation is allowed to use atomic operations on memory locations stronger than read/write only when there is contention. Moreover, this implementation guarantees that the operations that do not modify the object never return $\perp$ and use only read/write operations. This implementation is based on the primitive operation on memory locations Compare&Swap, whose computational power is the same as LL/SC.

k-Abortable objects This notion was recently introduced in [8]. A *k-abortable* object guarantees progress even under high contention, where "progress" means that $\perp$ cannot be returned by some operation invocations.

Roughly speaking an operation invoked by a process is allowed to abort only if it is concurrent with operations issued by k distinct processes and none of them returns $\perp$. This means that the k operations that entail the abort of another operation must succeed. It is easy to see that n-abortability is wait-freedom where any operation returns a non-$\perp$ result. A formal presentation can be found in [8].

A universal construction for k-abortable objects suited to the system model $\mathcal{CARW}_n[\text{LL/SC}]$ is presented in [8]. Differently from the construction for abortable objects presented in Figure 5, it is not a trivial construction. It uses an array of n memory locations $BOARD[1..n]$ used by the processes to store their last operations (they are the equivalent of the collect object $BOARD[1..n]$ used in Figure 3), an array of k memory locations $WINNERS[1..k]$ which contains the (up to k) "winning" operations, and another memory location $STATE$ (similar to the location $STATE$ used in Figure 3). All these memory locations are accessed with the LL/SC atomic operations. (We use the same identifiers as in Figure 3 to facilitate the understanding.)

The construction works as follows. After it has registered its operation in $BOARD[i]$, a process p_i tries to find an available entry in $WINNERS[1..k]$. If it succeeds, its operation will not abort; otherwise its operation will eventually abort. In all cases, i.e., whatever the fate of its own operation, the process p_i will help the winning operations to terminate. This construction is efficient in the sense that each operation terminates in $O(k)$ accesses to memory locations.

Let us observe that, as every k-abortable object can easily implement its k-lock-free counterpart, the previous universal construction for k-abortable objects is k-NB-compliant universal construction. Let us remember that, differently from its k-lock-free counterpart, no process can get stuck when a k-abortable object is used.)

5 From Operations on Memory Locations to Agreement Objects

5.1 Primitive operations versus objects

The previous universal constructions are based on hardware-provided atomic operations such as LL/SC. This operation, as all the hardware-provided synchronization operations (such as Test&Set or Compare&Swap) is uniform in the sense that they can be applied to any memory location [6, 22]. Hence the following natural questions come to mind:

- Is it possible to design a universal construction with other hardware-provided atomic operations such as Test&Set or Fetch&Add, initially designed to solve synchronization issues? Moreover, which synchronization atomic operations are equivalent (from the point of view of a universal construction)?
- Is it possible to generalize the concept of a universal construction to the coordinated construction of several objects with different progress conditions?

These questions are answered in this section.

5.2 A fundamental agreement object: consensus

Differently from a memory location which is only a sequence of bits accessed by hardware-provided atomic operations, the aim of an object is to provide its user with a high abstraction level (by hiding implementation details) and allow easier reasoning and proofs. An object is defined by a set of operations, and a specification which describes its correct behavior. The operations associated with an object are specific to it (i.e., due the very essence of the object concept, they are not uniform).

The consensus object The consensus object is the fundamental object associated with agreement problems. Introduced (in a different form) in the context of Byzantine synchronous message-passing systems [46], a consensus object provides the processes with a single operation denoted $\mathsf{propose}()$ that a process can invoke only once (one-shot object). When a process invokes $\mathsf{propose}(v)$, we say that it "proposes the value v". This operation returns a result. If a process returns value w, we say that it "decides w". In the context of process crash failures, the consensus object is defined by the following set of properties (let us remind that a correct process is a process that does not crash).

- Termination. If a correct process invokes propose(), it decides a value.
- Validity. A decided value is a proposed value.
- Agreement. No two processes decide different values.

A consensus object allows the processes to agree on the same value, and this value is not arbitrary: it was proposed by one of them. Hence, when considering a universal construction, consensus objects can be used by the processes to agree on the order in which their operations must be applied to the object that is built.

5.3 A simple consensus-based universal construction

A simple WF-compliant consensus-based universal construction is described in Figure 6. This construction, proposed in [30], is inspired from the state machine replication paradigm [43] and the consensus-based atomic broadcast algorithm presented in [18]. The reader will find a proof of it in [52]. Let O be the object that is built. As in Section 3, its sequential behavior is defined by a transition function $\delta()$.

Local variables A process p_i manages locally a copy of the object, denoted $state_i$, an array $sn_i[1..n]$ where $sn_i[j]$ denotes the sequence number of the last operation on O issued by p_j locally applied to $state_i$. The local variables $done_i$, res_i, $prop_i$, k_i, and $list_i$, are auxiliary variables whose meaning is clear from the context; $list_i$ is a list of pairs of (operation, process identity); $|list_i|$ is its size, and $list_i[r]$ is its r^{th} element; hence, $list_i[r].op$ is an object operation and $list_i[r].proc$ the process that issued it.

```
when $p_i$ invokes op($in$) do
    $done_i \leftarrow$ false; $BOARD[i] \leftarrow \langle \mathsf{op}(in), sn_i[i]+1 \rangle$;
    wait ($done_i$); return($res_i$).

Underlying local task $T$:     % background server task %
    while (true) do
        $prop_i \leftarrow \epsilon$;   % empty list %
        for $j \in \{1,\ldots,n\}$ do
            if $(BOARD[j].sn > sn_i[j])$ then
                append $(BOARD[j].op, j)$ to $prop_i$
            end if
        end for;
        if $(prop_i \neq \epsilon)$ then
            $k_i \leftarrow k_i + 1$;
            $list_i \leftarrow CONS[k_i]$.propose($prop_i$);
            for $r = 1$ to $|list_i|$ do
                $\langle state_i, res_i \rangle \leftarrow \delta(state_i, list_i[r].op)$;
                let $j = list_i[r].proc$; $sn_i[j] \leftarrow sn_i[j] + 1$;
                if $(i = j)$ then $done_i \leftarrow$ true end if
            end for
        end if
    end while.
```

Figure 6: A wait-free consensus-based universal construction (code for process p_i)

Shared Objects The shared memory contains the following objects.

- An array $BOARD[1..n]$ of single-writer/multi-reader atomic registers. Each entry is a pair such that the pair $\langle BOARD[j].op, BOARD[j].sn \rangle$ contains the last operation issued by p_j and its sequence number. Each $BOARD[j]$ is initialized to $\langle \perp, 0 \rangle$.
- An unbounded array $CONS[1..]$ of consensus objects.

Process behavior When a process p_i invokes an operation $\mathsf{op}(in)$on O, it registers it and its associated sequence number in $BOARD[i]$ (line 1). Then, it waits until the operation has been executed, and returns its result (line 2).

The array $BOARD$ constitutes the helping mechanism used by the background task of each process p_i. This task is made up two parts, which are repeated forever. First, p_i build a proposal $prop_i$, which includes the last operations (at most one per process) which not have been applied to the object O, from its local point of view (lines 4-9 and predicate of line 6). Then, if the sequence $prop_i$ is not empty, p_i proposes it to the next consensus instance $CONS[k_i]$ line 12). The resulting value $list_i$ is a sequence of operations proposed by a process to this consensus instance. Process p_i then applies this sequence of operations to its local copy $state_i$ of O (line 14), and updates accordingly its local array sn_i (line 15). If the operation that was applied is its own operation, p_i sets the Boolean $done_i$ to true (line 16), which will terminate its current invocation (line 2).

Bounded wait-freedom versus unbounded wait-freedom This construction ensures that the operations issued by the processes are wait-free, but does not guarantee that they are bounded-wait-free, namely, the number of steps (accesses to the shared memory) executed before an operation terminates is finite but not bounded. Consider a process p_i that issues an operation $\mathsf{op}()$, while $k1$ is the value of k_i. let and $k2 = k1 + \alpha$ be such that $\mathsf{op}()$ is output by the consensus instance $CONS[k2]$. The task T of p_i must execute α times the lines 4-18 in order to catch up the consensus instance $CONS[k2]$ and return the result produced by $\mathsf{op}()$. It is easy to see that the quantity $(k2 - k1)$ is always finite but cannot be bounded.

A bounded construction is described in [32]. Instead of requiring each process to manage a local copy of the object, O is kept in shared memory and is represented by a list of cells including an operation, the resulting state, the result produced by this operation, and a consensus object whose value is a pointer to the next cell. The last cell defines the current value of the object.

5.4 Consensus number and the consensus hierarchy

Consensus number of an object The notion of the *consensus number* of an object was introduced by M. Herlihy in [32]. Let us consider an object of type T (defined by a sequential specification). The *consensus number* of an object of type T is the greatest integer n such that it is possible to implement a consensus object in a system of n processes, with any number of atomic read/write registers and objects of type T. The consensus number is $+\infty$ if there is no largest n.

This notion allows us to answer the first question posed in Section 5.1, and this answer defines what is called the object *consensus hierarchy*. More precisely, it has been shown in [32] that:

- The consensus number of read/write registers is 1. It follows that all objects that can be built from read/write registers only (i.e., in $\mathcal{CARW}_n[\emptyset]$ without enrichment with additional operations) have consensus number 1. Snapshot objects [1, 5] and renaming objects [7, 16] are such objects).
- The consensus number of hardware operations such as Test&Set, Fetch&Add, Swap (exchange the values in a local register an a shared register), and a few others, have consensus number 2. This means that a universal construction can be built in $\mathcal{CARW}_2$[Test&Set] (i.e., in a system of two processes), but impossible in $\mathcal{CARW}_n$[Test&Set] for $n > 2$.
- Let a k-window read/write register be a register that stores only the sequence of the last k values which have been written, and whose read operation returns this sequence of at most k values. It is shown in [49] that the consensus number of a k-window is k.
- Finally, the consensus number of Compare&Swap, LL/SC, and a few others, is $+\infty$.

This infinite hierarchy is the *consensus hierarchy*. It provides us with a ranking of the power of synchronization objects and hardware provided synchronization operations in wait-free systems (i.e., systems where all, except one, processes may crash). As an example, if any number of processors may crash, this hierarchy states that a multicore with Test&Set is computationally less powerful than a multicore with LL/SC.

Consensus from several operations on memory locations The previous hierarchy considers that consensus must be built from read/write registers and objects of a given type T only. What can be done when several hardware operations which access the same memory locations are given?

As an example, let us consider the system model $\mathcal{CARW}_n$[Test&Set, Fetch&Add2] (defined in [22]) where Test&Set and Fetch&Add2 are two atomic operations defined as follows:

- Test&Set returns the value of the memory location, and sets it to 1 if it contained 0,
- Fetch&Add2 returns the value in the memory location and increases it by 2.

Each of these operations on memory locations has consensus number 2. The algorithm described in Figure 7 (due to F. Ellen, G. Gelashvili, N. Shavit, and L. Zhu, [22]) shows that a binary consensus object can be built in $\mathcal{CARW}_n$[Test&Set, Fetch&Add2], for any value of n. This means that the previous hierarchy collapses when object types defined by operations on memory locations can be used to implement consensus. Binary consensus means that only the values 0 and 1 can be proposed. This is not a problem as it is possible to build a multivalued consensus object from binary consensus objects (see [52]).

```
when $p_i$ invokes propose($v$) do
(1)  if $(v = 0)$ then $X$.fetch&add2();
(2)                   if ($X$ is odd) then return$(1)$ else return$(0)$ end if
(3)              else $x \leftarrow X$.test&set();
(4)                   if ($x$ is odd) $\vee$ ($x = 0$) then return$(1)$ else return$(0)$ end if
(5)  end if.
```

Figure 7: A wait-free consensus algorithm in $\mathcal{CARW}_n$[Test&Set, Fetch&Add2] (code for process p_i)

The internal representation of the binary consensus object is a single memory location X, initialized to 0. According to the value it proposes (0 or 1), a process executes the statements of lines 2-3 or the statements of lines 4-5. The value returned by the consensus object is sealed by the first atomic operation that is executed. It is 0 if the first operation on X is X.fetch&add2(), and 1 if first operation on X is X.test&set(). The reader can check that, if the first operation on X is fetch&add2(), X becomes and remains even forever. If it is test&set(), X becomes and remains odd forever. In the first case, only 0 can be decided, while in the second case, only 1 can be decided.

Power number The notion of the *power number* of an object type T (PN(T)) was introduced by G. Taubenfeld in [59]. It is the largest integer k such that it is possible to implement a k-obstruction-free consensus object for *any* number of processes, using any number of atomic read/write registers, and any number of objects of type T (the registers and the objects of type T being wait-free). If there is no such largest k, PN(T) = $+\infty$.

Hence, the power number of an object type T relates k-obstruction-freedom and wait-freedom, when objects of type T are used. Let CN(T) be the consensus number of the objects of type T. It is shown in [59] that CN(T) = PN(T). This result establishes a strong relation linking wait-freedom and k-obstruction-freedom. As noticed in [59], “the difficult part of the proof is to show that, for any $k \geq 1$, it is possible to implement a k-obstruction-free consensus algorithm for any number of processes, using only wait-free consensus objects for k processes and atomic read/write registers”.

5.5 Universal construction "1 among k"

k-Set agreement k-Set agreement (k-SA) was introduced by S. Chaudhuri [19]. It is a simple generalization of consensus. It is defined by the same validity and termination properties, and a weaker agreement property, namely, at most k different values can be decided by the processes. Hence, 1-set agreement is consensus. It is shown in [10, 37, 56] that it is impossible to build a k-set agreement object in $\mathcal{CARW}_n[\emptyset]$ when k or more processes may crash.

k-simultaneous consensus k-Simultaneous consensus (k-SC) was introduced in [3]. As consensus and k-SA, a k-SC object is a one-shot object that provides the processes with a single operation denoted $\mathsf{propose}()$. This operation takes an input parameter a vector of size k, whose each entry contains a value, and returns a pair $\langle x, v\rangle$. The input vector contains "proposed" values, and if $\langle x, v\rangle$ is the pair returned to the invoking process, this process "decides v, and this decision is associated with the consensus instance x", $1 \leq x \leq k$.

More precisely, the behavior of a k-SC object is defined by the following properties.

- Termination. If a correct process invokes $\mathsf{propose}()$, it decides a pair $\langle x, v\rangle$.
- Validity. If a process p_i decides the pair $\langle x, v\rangle$, we have $1 \leq x \leq k$, and the value v was proposed by a process in the entry x of its input vector parameter.
- Agreement. Let p_i be a process that decides the pair $\langle x, v\rangle$, and p_j be a process that decides the pair $\langle y, w\rangle$. We have $(x = y) \Rightarrow (v = w)$.

It is shown in [3] that k-SA and k-SC have the same computational power in the sense that a k-SA object can be built in $\mathcal{CARW}_n[k\text{-SC}]$, and a k-SC object can be built in $\mathcal{CARW}_n[k\text{-SA}]$. This equivalence is no longer true in asynchronous crash-prone message-passing systems, where k-SC is stronger than k-SA [12, 54].

Let $in_i[1..k]$ be the input parameter of a process p_i. An easy implementation of k-SC in $\mathcal{CARW}_n[\emptyset]$ enriched with k consensus objects $CONS[1..k]$ is as follows. For each x, $1 \leq x \leq k$, and in parallel, a process p_i proposes $in_i[x]$ to the consensus object $CONS[x]$. Let $CONS[y]$ be the first consensus object which returns a value v to p_i. Process p_i decides then the pair $\langle y, v\rangle$.

The notion of k-universality E. Gafni and R. Guerraoui investigated in [27] the following question: What does happens if, instead of consensus objects, we use k-SA (or equivalently k-SC) objects to design a universal construction?

They showed that it is then possible to design what they called a *k-universal construction*. Such a construction considers k objects (instead of only one) and guarantees that at least one of these objects progresses forever. Let GG denote the k-universal construction described in [27].

Adopt-commit object The GG construction relies on k-SC objects and adopt-commit (AC) objects. This object, introduced in [26], is a one-shot object which provides the processes with a single operation denoted $\mathsf{propose}()$, which takes a value as input parameter and returns a pair composed of a tag and a value. Its behavior is defined by the following properties.

- Validity.
  - Result domain. Any returned pair $\langle tag, v\rangle$ is such that (a) v has been proposed by a process and (b) $tag \in \{\texttt{commit}, \texttt{adopt}\}$.
  - No-conflicting values. If a process p_i invokes $\mathsf{propose}(v)$ and returns before any other process p_j has invoked $\mathsf{propose}(w)$ with $w \neq v$, only the pair $\langle \texttt{commit}, v\rangle$ can be returned.
- Agreement. If a process returns $\langle \texttt{commit}, v\rangle$, only the pairs $\langle \texttt{commit}, v\rangle$ or $\langle \texttt{adopt}, v\rangle$ can be returned by the other processes.

- Termination. An invocation of $\mathsf{propose}()$ by a correct process always terminates.

It follows from the "no-conflicting values" property that, if a single value v is proposed, only the pair $\langle \mathtt{commit}, v\rangle$ can be returned. Adopt-commit objects can be wait-free implemented in $\mathcal{CARW}_n[\emptyset]$ (e.g., [26, 52]). Hence, they provide processes with a higher abstraction level than read/write registers, but do not provide them with additional computational power.

A non-blocking k-universal construction (This section borrows text from [55]) The algorithm GG is based on local replication paradigm, namely, the only shared objects are the control objects $KSC[1..]$ (unbounded list of k-SC objects) and $AC[1..][1..k]$ (matrix of adopt-commit objects). Each process p_i manages a copy of every object m, denoted $state_i[m]$, which contains the last state of m as known by p_i. The invocation by p_i of $\delta(state_i[m], \mathsf{op})$ applies the operation $\mathsf{op}()$ to its local copy of object m. The construction consists in an infinite sequence of asynchronous rounds, locally denoted r_i at process p_i.

Each process manages the following local data structures.

- For each object m, $my\_list_i[m]$ defines the list of operations that p_i wants to apply to the object m. Moreover, $my\_list_i[m].\mathsf{first}()$ sets the read head to point to the first element of this list and returns its value; $my\_list_i[m].\mathsf{current}()$ returns the operation under the read head; finally, $my\_list_i[m].\mathsf{next}()$ advances the read head before returning the operation pointed to by the read head.
- For each object m, $oper_i[m]$, $ac\_op_i[m]$ are local variables which contain operations that p_i wants to apply object m (this list can be defined dynamically according to the algorithm executed by p_i); $tag_i[m]$ is used to contain a tag returned by an adopt-commit object concerning the object m.

The algorithm is presented in Figure 8. A process p_i first initializes its round number, and the local copy of each object. The array $oper_i[1..k]$ is such that $oper_i[m]$ contains the next operation that p_i wants to apply to m. When this is done, it enters an infinite loop, which constitutes the core of the construction. To simplify the presentation, and without loss of generality, we consider that all object operations are different (this can be easily realized with sequence numbers and process identities). Moreover, we also do not consider the result returned by each operation.

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 $\langle obj, op\rangle$; op is an operation proposed by some process for the object obj (line 2). Process p_i then invokes the adopt-commit object $AC[r][obj]$ to which it proposes the operation op output by $KSC[r]$ for the object obj (line 3). Finally, for all the other objects $m \neq obj$, 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 operations defined by the vector of pairs output by the adopt-commit objects $AC[r][1..k]$ are saved in the vectors $tag_i[1..k]$;and $ac\_op_i[1..k]$, respectively. The aim of these lines, realized by the objects $KSC[r]$ and $AC[r][1..m]$is to implement a filtering mechanism such that (a) for each object, at most one operation can be be committed, and (b) there is at least one object for which an operation is committed at some process. This filtering mechanism is explained separately below.

After the execution lines 2-4, for $1 \leq m \leq k$, $\langle tag_i[m], ac\_op_i[m]\rangle$ contains the operation that p_i has to consider for the object 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 $state_i[m]$ (lines 6-8). Then, the predicate of line 9 ensures that no operation invocation is applied twice on the same object (this line is missing in [27]). If $tag_i[m] = \mathtt{adopt}$, p_i adopts $ac\_op_i[m]$ as its next proposal for the object m (lines 10-11). Otherwise, $tag_i[m] = \mathtt{commit}$. In this case p_i first applies $ac\_op_i[m]$ to its local copy $state_i[m]$ (line 12). Then, if $ac\_op_i[m]$ was an operation it has issued, p_i computes its next operation $oper_i[m]$ on the object m (lines 13-16).

```
$r_i \leftarrow 0$;
for each $m \in \{1, ..., k\}$ do
        $state_i[m] \leftarrow$ initial state of the object $m$; $oper_i[m] \leftarrow my\_list_i[m]$.first()
end for.

repeat forever
(1)   $r_i \leftarrow r_i + 1$;
(2)   $\langle obj, op\rangle \leftarrow KSC[r_i]$.propose($oper_i[1..k]$);
(3)   $(tag_i[obj], ac\_op_i[obj]) \leftarrow AC[r_i][obj]$.propose($op$);
(4)   for each $m \in \{1, ..., k\} \setminus \{obj\}$ do
              $(tag_i[m], ac\_op_i[m]) \leftarrow AC[r_i][m]$.propose($oper_i[m]$) end for;
(5)   for each $m \in \{1, ..., k\}$ do
(6)           if ($ac\_op_i[m]$ is marked "to_be_executed_after" $oper_i[m]$)
(7)              then $state_i[m].\delta(state_i[m], oper_i[m])$
(8)           end if;
(9)           if ($oper_i[m]$ is not marked "to_be_executed_after″ $ac\_op_i[m]$)
(10)             then  if ($tag_i[m] =$ adopt)
(11)                      then $oper_i[m] \leftarrow ac\_op_i[m]$
(12)                      else  $state_i[m] \leftarrow \delta(state_i[m], ac\_op_i[m])$; % $tag_i[m] =$ commit %
(13)                            if $ac\_op_i[m] = my\_list_i[m]$.current()
(14)                               then $oper_i[m] \leftarrow my\_list_i[m]$.next()
(15)                               else  $oper_i[m] \leftarrow my\_list_i[m]$.current()
(16)                            end if;
(17)                            mark $oper_i[m]$ "to_be_executed_after" $ac\_op_i[m]$
(18)                   end if
(19)          end if
(20)  end for
end repeat.
```

Figure 8: Non-blocking k-universal construction (code of p_i)

As explained in [27], the use of a naive strategy to update local copies of the objects, makes possible the following bad scenario. During a round r, a process p_1 executes an operation op1 on its copy of the object $m1$, while a process p_2 executes a operation op2 on a different object $m2$. Then, during round $r+1$, p_1 executes a operation op3 on the object $m2$ without having executed first op2 on its copy of $m2$. This bad behavior is prevented from occurring by a combined used of adopt-commit objects and an appropriate marking mechanism. When a process p_i applies an operation op() to its local copy of an object m, it has necessarily received the pair $\langle$commit, op()$\rangle$ from the adopt-commit object associated with the current round, and consequently the other processes have received $\langle$commit, op()$\rangle$ or $\langle$adopt, op()$\rangle$. The process p_i attaches then to its next operation for the object m (which is denoted $oper_i[m]$) the indication that $oper_i[m]$ has to be applied to m after op() so that no process executes $oper_i[m]$ without having previously executed op(). Hence, to prevent the bad behavior previously described, a process p_i attaches to $oper_i[m]$ (line 17) the fact that this operation cannot be applied to any copy of the object m before the operation $ac\_op_i[m]$.

Al already indicated, this k-universal construction ensures that at least one process progresses forever (non-blocking progress condition), and at least one object progresses forever.

Why at least one object operation is committed at every round It was claimed above that the "filtering mechanism" realized by lines 2-4 ensures that at least one operation is committed at every round. We prove here this claim. Figure 9 illustrates the associated reasoning.

After a process p_{i1} obtained a pair $\langle obj1, op1\rangle$ from its invocation $KSC[r]$.propose($oper_i[1..k]$) at line 2, it invokes $AC[r][obj1]$.propose($op1$) at line 3, and only then it invokes $AC[r][obj]$.propose($op1$) for each object $obj \neq obj1$ at line 4. If its invocation of $AC[r][obj1]$.propose($op1$) at line 3 returns $\langle$commit, $-\rangle$, the claim follows.

Hence, let us assume that the invocation of $AC[r][obj1]$.propose($op1$) by p_{i1} returns $\langle \texttt{adopt}, - \rangle$. It follows from the "non-conflicting" property of the AC object $AC[r][obj1]$ that another process p_{i2} has necessarily invoked $AC[r][obj1]$.propose(op') with $op' \neq op1$; moreover this invocation by p_{i2} was issued at line 4 (if both p_{i1} and p_{i2} had invoked $AC[r][obj1]$.propose() at line 3, due to agreement property of $AC[r][obj1]$, they would have obtained the same pair from this object at line 3, and consequently p_{i2} could not have prevented p_{i1} from obtaining $\langle \texttt{commit}, - \rangle$ from the AC object $AC[r][obj1]$ at line 3). If follows that p_{i2} started line 4 before p_{i1} terminated line 3. The invocation by p_{i2} at line 3 of $AC[r][-]$ involved some object $obj2$ obtained by p_{i2} at line 2, and we necessarily have $obj2 \neq obj1$).

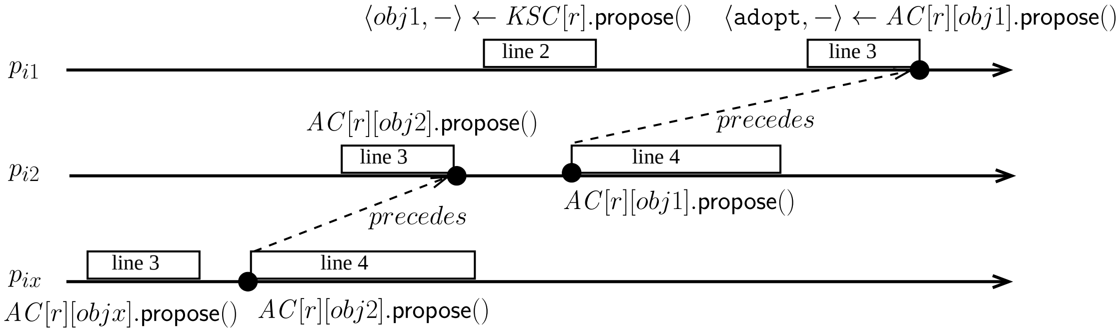

Figure 9: Net effect of the k-SC and CA objects used at lines 2-4 of round r

If the invocation of $AC[r][obj2]$.propose() returns $\langle \texttt{commit}, - \rangle$, the claim follows. Otherwise, due to the agreement property of $AC[r][obj2]$, there is a process p_{i3}, different from p_{i1} and p_{i2}, such that the execution pattern between $p_{i3} \neq p_{i2}$ is the same as the previous pattern between $p_{i2} \neq p_{i1}$, etc. The claim then follows by induction and the fact that there is finite number of processes.

5.6 Ultimate universal construction "ℓ among k"

The previous NB-compliant k-universal construction ensures that at least one object progresses forever, and one process progresses forever. Hence, the natural question: Is it possible to design a universal construction in which at least ℓ objects progress forever, where $1 \leq \ell \leq k$, and all correct processes progress forever (wait-freedom progress condition).

Such a very general universal construction was proposed by M. Raynal, J. Stainer, and G. Taubenfeld in [55]. It rests on an extension of the k-SC object called (k, ℓ)-simultaneous consensus.

(k, ℓ)-simultaneous consensus Let $\ell \in \{1, ..., k\}$. A (k, ℓ)-SC object is a k-SC object (see Section 5.5) where instead of a single pair $\langle x, v \rangle$, the operation propose() returns a set of exactly ℓ pairs $\{\langle x_1, v_1 \rangle, ..., \langle x_\ell, v_\ell \rangle\}$, such that all the pairs differ in their first component.

It is easy to see that $(k, 1)$-SC object is a k-SC object (and consequently a k-SA object). Moreover, a (k, k)-SC object is a consensus object. It is also easy to see that a (k, k)-SC object is a consensus object. For $k > 1$, a (k, ℓ)-SC object is weaker than a $(k, \ell + 1)$-SC object.

(k, ℓ)-Universal construction The (k, ℓ)-universal construction presented in [55] borrows the lines 1-4 of Figure 8, in which k-SC objects are replaced by (k, ℓ)-SC objects. All the rest of the construction, which is built incrementally, is based on a different approach. A non-blocking k-universal construction is first described, and then enriched step by step to obtain the final WF-compliant (k, ℓ)-universal construction. Its noteworthy features 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 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 stop in the same state (at the non-crashed processes).
- The construction is *contention-aware*. This means that the overhead introduced by using operations on memory locations other than atomic read/write registers is eliminated when there is no contention during the execution of an object operation. In the absence of contention, a process completes its operations by accessing only read/write registers.
- 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.
- Last but least, it is shown in [55] that (k, ℓ)-simultaneous consensus objects are necessary and sufficient to implement a (k, ℓ)-universal construction, i.e. to ensure that at least ℓ among k objects progress forever while guaranteeing the wait-freedom progress condition to the processes. Relations between $(k, k-p)$-SC objects and $(p+1)$-set agreement objects for $0 \leq p < k$ are also investigated in [55].

6 Universal Construction vs Software Transactional Memory

A universal construction is on the distributed implementation of concurrent objects defined by a sequential specification. The concept of a *software transactional memory* (STM), introduced in [35], and later refined in [57], is different. Its aim is to provide the programmers with a language construct (called *transaction*) that discharges them from the management of synchronization issues. In this way, a programmer can concentrate his efforts on which parts of processes have to be executed atomically and not on the way atomicity is realized. This last issue is then the job of the underlying STM system. Among others, main differences between universal constructions and STM systems are the following.

- Object operations are defined a priori (statically), and the universal construction knows them. Differently, the transactions are defined dynamically, and the STM system has no priori knowledge of their content and their effects.

  Let us also notice that, despite the fact they have the same name, database transactions [28] and STM transactions are not the same. Database transactions are constrained in the sense that they are the result of a queries expressed in a given formalism. Differently, STM transactions can be any piece of code produced by a programmer, which must be executed atomically. Moreover, usually the code of the STM transactions is not known by the STM system.
- The consistency condition of concurrent objects (captured at run-time by linearizability [38]) and the consistency conditions of STM systems (e.g., opacity [29], virtual world consistency [40], or TMS1 [20]) are different. Among other points, this come from the fact that any two transactions are a priori independent.
- Due to their very nature, universal constructions consider failure-prone systems. Differently, some STMs address failure-free systems while others address failure-prone systems.

7 Conclusion

The aim of this article was to be a guided visit to universal constructions in asynchronous crash-prone systems, where the processes communicate through a shared memory. As announced in the introduction, its ambition is not to be an exhaustive catalog of the numerous universal constructions proposed so far, but a relatively easy to understand introduction to the "universal construction" problem and the important concepts, objects, and approaches, which constitute the foundations of the associated algorithms.

To this end, the article has first presented a simple construction based on hardware operations on memory locations, namely the LL/SC pair of operations. It then moved from hardware-provided operations to agreement objects, and presented a simple consensus-based universal construction. Finally, the article considered the case where the aim is not to address the construction of a single object, but the coordinated construction of several objects. It is important to realize that, if not all the objects which are built are required to progress forever, hardware operations such as LL/SC or Compare&Swap are stronger than necessary to build universal constructions.

As a final remark, let us notice that OB-compliant (obstruction-free) universal constructions do not require to enrich the system with the additional computational power provided by instructions such as LL/SC or agreement objects, i.e., they can be done in the basic system model $\mathcal{CARW}[\emptyset]$. This remains true even if the processes are anonymous. The algorithms presented in [11] build a consensus object and a repeated consensus object respectively, in such an asynchronous crash-prone anonymous read/write system with only n read/write atomic registers, which we conjecture to be optimal (it is proved in [63] that at least $(n-1)$ registers are necessary).

References

[1] Afek Y., Attiya H., Dolev D., Gafni E., Merritt M., and Shavit N., Atomic snapshots of shared memory. *Journal of the ACM*, 40(4):873-890 (1993)

[2] Afek Y., Dauber D., and Touitou D., Wait-free made fast. *Proc. 27th ACM Symposium on Theory of Computing (STOC'95)*, ACM Press, pp. 538-547 (1995)

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

[4] Aguilera M.K., Frolund S., Hadzilacos V., Horn S.L., and Toueg S., Abortable and query-abortable objects and their efficient implementation. *Proc. 26th ACM Symposium on Principles of Distributed Computing (PODC'07)*, ACM Press, pp. 23-32 (2007)

[5] Anderson J.H., Multi-writer composite registers. *Distributed Computing*, 7(4):175-195 (1994)

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

[7] Attiya H., Bar-Noy A., Dolev D., Peleg D., and Reischuk R., Renaming in an asynchronous environment. *Journal of the ACM*, 37(3):524-548 (1990)

[8] Ben-David N., Cheng Chan D.Y., Hadzilacos V. and Toueg S., k-Abortable objects: progress under high contention. *Proc. 30th Int'l Symposium on Distributed Computing (DISC'16)*, Springer LNCS 9888, pp. 298-312 (2016)

[9] Bartlett K. A., Scantlebury S. A., and Wilkinson P. T., A note on reliable full-duplex transmission over half-duplex links. *Communications of the ACM*, 12(5):260-261 (1969)

[10] 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)

[11] Bouzid Z., Raynal M., and Sutra P., Anonymous obstruction-free (n, k)-set agreement with $(n - k + 1)$ atomic read/write registers. *Proc. 19th Int'l Conference On Principles Of Distributed Systems (OPODIS'15)*, Leibniz Int'l Proceedings in Informatics, LIPICS 46, Article 18:1-17 (2015)

[12] 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)

[13] Brinch Hansen, P., *The origin of concurrent programming*. Springer, 534 pages, ISBN 0-387-95401-5 (2002)

[14] Bushkov V. and Guerraoui G., Safety-liveness exclusion in distributed computing. *Proc. 34th ACM Symposium on Principles of Distributed Computing (PODC'15)*, ACM Press, pp. 227-236 (2015)

[15] Capdevielle Cl., Johnen C., and Milani A., Solo-fast universal constructions for deterministic abortable objects. *Proc. 28th Int'l Symposium on Distributed Computing (DISC'14)*, Springer LNCS 8784, pp. 288-302 (2014)

[16] Castañeda A., Rajsbaum S., and Raynal M., The renaming problem in shared memory systems: an introduction. *Elsevier Computer Science Review*, 5:229-251 (2011)

[17] Censor-Hillel K., Petrank E., and Timnat S., Help! *Proc. 34th Symposium on Principles of Distributed Computing (PODC'15)*, ACM Press, pp. 241-250 (2015)

[18] Chandra T.D. and Toueg S., Unreliable failure detectors for reliable distributed systems. *Journal of the ACM*, 43(2):225-267 (1996)

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

[20] Doherty S., Groves L., Luchangco V., and Moir M., Towards formally specifying and verifying transactional memory. *Formal Aspects of Computing*, 25:769âĂŞ799 (2013)

[21] Ellen F., Fatourou P., Kosmas E., Milani A., and Travers C., Universal constructions that ensure disjoint-access parallelism and wait-freedom. *Distributed Computing*, 29:251-277 (2016)

[22] Ellen F., Gelashvili G., Shavit N. and Zhu L., A complexity-based hierarchy for multiprocessor synchronization (Extended abstract). *Proc. 35th ACM Symposium on Principles of Distributed Computing (PODC'16)*, ACM Press, pp. 289-298 (2016)

[23] Fatourou P. and Kallimanis N.D., The RedBlue adaptive universal constructions. *Proc. 23rd Symposium on Distributed Computing (DISC'09)*, Springer LNCS 5805, pp. 127-141 (2009)

[24] Fatourou P. and Kallimanis N.D., Highly-efficient wait-free synchronization. *Theory of Computing Systems*, 55:475-520 (2014)

[25] 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)

[26] 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)

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

[28] Gray J., Notes on database operating systems. *Advanced course on Operating Systems*, Springer LNCS 60, pp. 393-481 (1978)

[29] Guerraoui R. and Kapalka M., On the correctness of transactional memory. *Proc. 3rd ACM Symposium on Principles an Practice of Parallel Programming (PPOPP'03)*, ACM Press, pp. 175-184 (2008)

[30] Guerraoui R. and Raynal M., A universal construction for wait-free objects. *Proc. Workshop on Foundations of Fault-Tolerant Distributed Computing (FOFDC 2007)*, Computer Society Press, pp. 959-966 (2007)

[31] Hadzilacos V. and Toueg S., On deterministic abortable objects. *Proc. 35th ACM symposium on Principles of Distributed Computing (PODC'13)*, ACM Press, pp. 4-12 (2013)

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

[33] Herlihy M.P., A methodology for implementing highly concurrent data objects. *ACM Transactions on Programming Languages and Systems*, 15(5):745-770 (1993)

[34] 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)

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

[36] Herlihy M., Rajsbaum S., and Raynal M., Power and limits of distributed computing shared memory models. *Theoretical Computer Science*, 509:3-24 (2013)

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

[38] 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)

[39] Imbs D. and Raynal M., Help when needed, but no more: efficient read/write partial snapshot. *Journal of Parallel and Distributed Computing*, 72(1):1-13 (2012)

[40] Imbs D. and Raynal M., Virtual world consistency: A condition for STM systems (with a versatile protocol with invisible read operations). *Theoretical Computer Science*, 444:113-127 (2012)

[41] Imbs D., Raynal M., and Taubenfeld G., On asymmetric progress conditions. *Proc. 29th ACM Symposium on Principles of Distributed Computing (PODC'10)*, ACM Press, pp. 55-64 (2010)

[42] Kramer S. N., *History begins at Sumer: thirty-nine firsts in man's recorded history*. University of Pennsylvania Press, 416 pages, ISBN 978-0-8122-1276-1 (1956)

[43] Lamport L., Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558-565 (1978)

[44] Lamport L., On interprocess communication, Part I: basic formalism. *Distributed Computing*, 1(2):77-85 (1986)

[45] Lamport L., Fast mutual exclusion. *ACM Transactions on Computer Systems*, 5(1):1-11 (1987)

[46] Lamport L., Shostack R. and Pease M., The Byzantine generals problem. *ACM Transactions on Programming Languages and Systems*, 4(3)-382-401 (1982)

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

[48] Lynch W. C., Reliable full-duplex file transmission over half-duplex telephone lines. *Communications of the ACM*, 11(6):407-410 (1968)

[49] Mostéfaoui A., Perrin M., and Raynal M., A simple object that spans the whole consensus hierarchy. *Submitted to publication*, (2016)

[50] Neugebauer O. E., *The exact sciences in Antiquity.* Princeton University Press (1952); 2nd edition: Brown University Press, (1957); Reprint: Dover publications (1969)

[51] Post E. L., Formal reductions of the general combinatorial decision problem. *American Journal of Mathematics*, 65 (2):197-215 (1943)

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

[53] Raynal M., Concurrent systems: hybrid object implementations and abortable objects. *Proc. 21th Int'l European Parallel Computing Conference (EUROPAR'15)*, Springer LNCS 9233, pp. 3-15 (2015)

[54] 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)

[55] Raynal M., Stainer J., and Taubenfeld G., Distributed universality. *Algorithmica*, 76(2):502-535 (2016)

[56] 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)

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

[58] Taubenfeld G., *Synchronization algorithms and concurrent programming*. 423 pages, Pearson Education/Prentice Hall, ISBN 0-131-97259-6 (2006)

[59] 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)

[60] Taubenfeld G., The computational structure of progress conditions. *Proc. 24th Int'l Symposium on Distributed Computing (DISC'10)*, Springer LNCS 6343, pp. 221-235 (2010)

[61] Turing A. M., On computable numbers with an application to the Entscheidungsproblem. *Proc. of the London Mathematical Society*, 42:230-265 (1936)

[62] Wantzel P. L., Recherches sur les moyens de reconnaître si un problème de géométrie peut se résoudre avec la règle et le compas, *Journal de mathématiques pures et appliquées*, 1(2):366-372 (1837)

[63] Zhu L., A tight space bound for consensus. *Proc. 48th ACM Symposium on Theory of Computing (STOC'16)*, ACM Press, pp. 345-350 (2016)