Solo-fast Universal Constructions for Deterministic Abortable Objects*

Claire Capdevielle, Colette Johnen, and Alessia Milani

Univ. Bordeaux, LaBRI, UMR 5800, F-33400 Talence, France

Abstract. In this paper we study efficient implementations for deterministic abortable objects. Deterministic abortable objects behave like ordinary objects when accessed sequentially, but they may return a special response *abort* to indicate that the operation failed (and did not take effect) when there is contention.
It is impossible to implement deterministic abortable objects only with read/write registers [4]. Thus, we study *solo-fast* implementations. These implementations use stronger synchronization primitives, e.g., CAS, only when there is contention. We consider interval contention.
We present a non-trivial solo-fast universal construction for deterministic abortable objects. A universal construction is a method for obtaining a concurrent implementation of any object from its sequential code. The construction is *non-trivial* since in the resulting implementation a failed process can cause only a finite number of operations to abort. Our construction guarantees that operations that do not modify the object always return a legal response and do not use CAS. Moreover in case of contention, at least one writing operation succeeds. We prove that our construction has asymptotically optimal space complexity for objects whose size is constant.

Keywords: Concurrent programming, abortable object, universal construction, solo-fast implementation, lower bound, space complexity, wait-freedom

1 Introduction

With the raise of multicore and many core machines efficient concurrent programming is a major challenge. Linearizable shared objects are central in concurrent programming ; They provide a convenient abstraction to simplify the design of concurrent programs. But implementing them is complex and expensive when strong progress conditions are required, e.g. wait-freedom (every process completes its operations in a finite number of steps) [10]. The complexity originates in executions where processes execute concurrent operations. Obstruction-freedom was proposed to circumvent this difficulty by allowing an operation to never return in case of contention [11]. This separation between correctness and progress let devise simpler and more efficient algorithms. In fact any obstruction free object can be implemented using only read/write registers.

On the other hand, as pointed out by Attiya et al., [4], ideally shared objects should always return the control, and when this happens the caller should know if the operation took place or not. This behavior is formalized in the notion of deterministic abortable object proposed by Hadzilacos and Toueg [9]. A deterministic abortable object ensures that if several processes contend to operate on it, it may return a special response *abort* to indicate that the operation failed. And it assures that an operation that aborts does not take effect. Operations that do not abort return a response which is legal w.r.t. the sequential specification of the object.

In this paper we study efficient implementations for deterministic abortable objects. Attiya et al. proved that it is impossible to implement deterministic abortable objects only with read/write registers, [4]. Thus, we study implementations that use only read/write registers when there is no contention and use stronger synchronization primitives, e.g., CAS, when contention occurs. These implementations are called *solo-fast* and are expected to take advantage of the fact that in practice contention is rare.

The notion of solo-fast was defined in [4] for *step contention* : There is step contention when the steps of a process are interleaved with the steps of another process. In the same paper, they prove a linear lower bound on the space complexity of solo-fast implementations of obstruction-free objects. This result also holds for deterministic abortable objects.

* This work was partially supported by the ANR project `Displexity`.

We consider an asynchronous shared-memory system where processes communicate through linearizable shared objects and can fail by crashing, i.e. ; a process can stop taking steps while executing an operation. In this model, we study the possibility that deterministic abortable objects can be implemented efficiently if a process is allowed to use strong synchronization primitives even in absence of step contention, provided that its operation is concurrent with another one. This notion of contention is called *interval contention* [1]. Step contention implies interval contention, the converse is not true. To avoid the situations in which a crashed process can prevent other processes to terminate, we consider only implementations where a crashed process can cause only a finite number of concurrent operations to abort. This property, called *non-triviality*, is formally defined in [2].

First we prove a linear lower bound on the space complexity of solo-fast implementations of abortable objects for our weaker notion of solo-fast. To prove our result we adapt the notion of pertubable object presented in [14] to abortable objects. We prove that a k-CAS abortable register is perturbable according to our definition, and, we prove the lower bound in a way similar to the proof in [4].

Then, we present a solo-fast universal construction for deterministic abortable objects. A *universal construction* [10] is a methodology for automatically transform any sequential object in a concurrent one. An implementation resulting from our universal construction is solo-fast and has asymptotically optimal space complexity if the implemented object has constant size. Our algorithm guarantees that operations that do not modify the object always return a legal response. Also in case of contention, at least one writing operation succeeds to modify the object. In particular, writing operations are applied one at the time. Each process makes a local copy of the object and computes the new state locally. We associate a sequence number to each state. A process that wants to modify the ith state has to compete to win the $i+1$th sequence number. A process that does not experience contention uses only read/write registers, while a CAS register is used in case of contention to decide the new state. It may happen that (at most) one process p behaves as if it was running solo, while other processes were competing for the same sequence number. In this case, we use a lightweight helping mechanism to avoid inconsistency : any other process acquires the state proposed by p as its new state. If it succeeds to apply it, it notifies the process p that its state has been applied. Then the helping process aborts. We ensure that if a process crashes while executing an operation, then it can cause at most two operations per process to abort. Our construction uses $O(n)$ read/write registers and $n+1$ CAS registers. Also it keeps at most $2n+1$ versions of the object.

Related work. Attiya et al. were the first to propose the idea of shared objects that in case of contention return a fail response [4]. Few variants of these objects have been proposed [4, 2, 9]. The ones proposed in [4, 2] differ from deterministic abortable objects in the fact that when a fail response is returned the caller does not know if the operation took place or not.

A universal construction for deterministic abortable objects is presented in [9]. This construction is derived from the universal construction presented in [10] and can be easily transformed into solo-fast by using the solo-fast consensus object proposed in [4]. This construction has unbounded space complexity, since it stores all the operations performed on the object. Also operations that only read the state of the object modify the representation of the implemented object and may fail by returning abort.

Several universal constructions have been proposed for ordinary wait-free concurrent objects. A good summary can be found in [5]. These constructions could be transformed in solo-fast by replacing the strong synchronization primitives they use with their solo-fast counterpart. To the best of our knowledge no solo-fast LL/SC or CAS register exist. Luchangco et al. presented a fast-CAS register [15] whose implementation ensures that no strong synchronization primitive is used in execution without contention. But, in case of contention, concurrent operations can leave the system in a state such that a successive operation will use strong synchronization primitives even if running solo. So, their implementation is not solo-fast. Even using this solo-fast consensus object Attiya at el, which has $\Theta(n)$ space complexity, we cannot easily modify existing universal constructions to make them solo-fast for abortable objects while ensuring all the good properties of our solution.

Abortable objects behave similarly to transactional memory, [12]. Transactional memory enables processes to synchronize via in-memory transactions. A transaction can encapsulate any piece of sequential code. This generality costs a greater overhead as compared to abortable objects. Also transactional memory is not aware of the sequential code embedded in a transaction. A hybrid approach between transactional memory and universal constructions has been presented by Crain *et al.* [6]. Their solution assumes that no failures occur. In addition they use a linked list to store all committed transactions. Thus, their solution has unbounded space complexity. Finally, our algorithm ensures *multi-version per-*

missiveness and *strong progressiveness* proposed for transactional memory respectively in [16] and in [8] when conflicts are at the granularity of the entire implemented object.

Paper organization. In Section 2 we present our model and preliminaries. In Section 3 we prove the linear lower bound on the space complexity of solo-fast implementations of obstruction-free objects. In Section 4 we present our solo-fast universal construction. Finally, the proofs of the construction are given in Section 5.

2 Preliminaries

We consider an asynchronous shared memory system, in which n processes $p_1 \dots p_n$ communicate through shared objects, such as read/write registers and CAS objects. Every object has a type that is defined by a quadruple (Q, O, R, Δ), where Q is a set of states, O is a set of invocations, R is a set of responses, and $\Delta \subseteq Q \times O \times Q \times R$ is the sequential specification of the type. A tuple (s, op, s', res) in Δ means that if type T is in state s when $op \in O$ is invoked, then T can change its state to s' and return the response res.

For each type $T = (Q, O, R, \Delta)$, we consider the deterministic abortable counterpart of T as defined in [9] and denoted T^{da}. T^{da} is equal to $(Q, O, R^{da}, \Delta^{da})$ where $R^{da} = R \cup \{\perp\}$ for some $\perp \notin R$, and, for every tuple (s, op, s', res) in Δ, the sequential specification Δ^{da} contains the following two tuples: (s, op, s', res) and $(s, op, s, \perp)$. These two tuples of Δ^{da} correspond to op completing normally, and op aborting without taking effect.

A universal construction is a method to transform any sequential object into a linearizable concurrent object. It consists in a method which takes as input the sequential code of an operation and its arguments. The algorithm that implements this method is a sequence of operations on shared objects provided by the system, called *base objects*. To avoid confusion between the base objects and the implemented ones, we reserve the term operation for the objects being implemented and we call *primitives* the operations on base objects. We say that an operation of an implemented object is performed and that a primitive is applied to a base object.

In the following, we consider that for any given object o the set of operations to access it is either *historyless* or not. Let o be a base object that supports two operations f and f'. Following [7], we say that f overwrites f' on o, if starting from any value v of o, applying f' and then f results in the same value as applying just f, using the same input parameters (if any) in both cases. A set of primitives is called *historyless* if all the primitives in the set that may change the state of the object overwrite each other; we also require that each such operation overwrites itself.

A step of a process consists of a primitive applied to a base object and possibly some local computation. A configuration specifies the value of each base object and the state of each process at some point in time. In an *initial configuration*, all base objects have their initial values and all processes are in their initial states. An *execution* is a (possibly infinite) sequence $C_i, \phi_i, C_{i+1}, \phi_{i+1}, \dots, \phi_{j-1}, C_j$ of alternating configurations (C_k) and steps (ϕ_k), where the application of ϕ_k to configuration C_k results in configuration C_{k+1}, for each $i \leq k < j$. For any finite execution α and any execution α', the execution $\alpha\alpha'$ is the concatenation of α and α'; in this case α' is called an extension of α. An execution α is q-free if no step in α is applied by the process q.

The *execution interval* of an operation starts with an invocation and terminates when a response is returned. An invocation without a matching response is a *pending* operation. Two operations op and op' are *concurrent* in a execution α, if they are both pending in some finite prefix of α. This implies that their intervals overlap. An operation op *precedes* an operation op' in α if the response of op precedes the invocation of op' in α. An operation experiences *interval contention* in an execution α if it is concurrent with at least another operation in α.

Processes may experience *crash failures*. For any given execution α, if a process p does not fail in α, we say that p is *correct* in α.

Properties of the implemented object. We consider universal construction that guarantees that all implementations resulting by their application are wait-free [10], linearizable [13], *non-trivial* and *non-trivial solo-fast*. *Wait-free* implementations ensure that in every execution, each correct process completes its operation in a finite number of steps. *Linearizability* ensures that for every execution α

and for every operation that completes and some of the uncompleted operations in α, there is some point within the execution interval of the operation called its linearization point, such that the response returned by the operation in α is the same as the response it would return if all these operations were executed serially in the order determined by their linearization points.

Informally, an implementation of an object is *non-trivial* if for every execution α every operation that aborts is concurrent with some other operation in α, and an operation that remains incomplete does not cause infinitely many other operations to abort. A more formal definition can be found in [2].

Finally, an implementation is said *non-trivial solo-fast* if for each execution α a process p applies some no historyless primitives during performing an instance of operation *op*, only if *op* is concurrent with some other operation in α and an operation that remains incomplete does not justify the application of no historyless primitives by infinitely many other operations.

3 Lower Bound

In the following we adapt the definition of perturbable objects presented in [3] and originally proposed in [14] to deterministic abortable objects.

Definition 1. *A deterministic abortable object O is perturbable for n processes, if for every linearizable and non-trivial implementation of O there is an operation instance op_n by process p_n, such that for any p_n-free execution $\alpha\lambda$ where no process applies more than a single event in λ and for some process $p_l \neq p_n$ that applies no event in λ, there is an extension of α, γ,consisting of events by p_l, such that the first response $res \neq \perp$, that p_n returns when repeatedly performing op_n by itself after $\alpha\lambda$ is different from the first response $res' \neq \perp$ it returns when repeatedly performing op_n by itself after $\alpha\gamma\lambda$.*

By adjusting the proof of Lemma 4.7 in [14], we prove that the set of deterministic abortable objects which are perturbable is not empty. In particular, we prove that the k-valued deterministic abortable CAS is perturbable.

A k-valued deterministic abortable CAS is the type (Q, O, R, Δ), where $Q = \{1, 2, .., k\}$, $O = \{Read, CAS(u, v)$ with $u, v \in \{1, 2, .., k\}\}$, $R = \{1, 2, .., k\} \cup \{true, false, \perp\}$ and $\forall s, u, v \in \{1, 2, .., k\}$ $\Delta = \{(s, Read, s, s)\} \cup \{(s, CAS(s, v), v, true)\} \cup \{(s, CAS(u, v), s, false)$ with $u \neq s\} \cup \{(s, Read, s, \perp)\} \cup \{(s, CAS(u, v), s, \perp)\}$.

Lemma 1. *For all $k \geq n$, k-valued deterministic abortable CAS object is perturbable for n processes for any initial state.*

Proof. Consider any linearizable and non-trivial implementation of a k-valued deterministic abortable CAS object O, initialized to any value and shared by processes $p_1, ..., p_n$. Let α and λ be any p_n-free execution where no process applies more than a single event in λ.

Let p_l be a process that applied no event in λ. If p_l have a pending operation in α let γ' be an extension of α that completes the pending operation. Let P be the set of processes that have a pending operation on O at the end of $\alpha\gamma'$, then $P \subseteq \{p_1, ..., p_{n-1}\} - \{p_l\}$.

Let Q be the set of processes that initiate a new operation on O during the execution λ. Since p_l applies no event in λ and λ is a p_n-free execution, then $Q \subseteq \{p_1, ..., p_{n-1}\} - \{p_l\}$. Moreover, if a process have a pending operation in $\alpha\gamma'$ it cannot initiate a new operation in λ because no process applies more than a single event in λ. Thus we have $P \cap Q = \emptyset$. Since $P, Q \subseteq \{p_1, ..., p_{n-1}\} - \{p_l\}$, we have $|P| + |Q| \leq n - 2$. Let V be the set of all v such that a $CAS(v, -)$ operation on O is either pending in $\alpha\gamma'$ or initiated in λ. From above $|V| \leq n - 2$.

Let res be the value returned by the first response which is not $\perp$ after that p_n performs a sequence of read operations after $\alpha\lambda$. Because of the non-triviality this value exists.

Let $w \in \{1, 2, .., n\}$ be such that $w \notin V$ and $w \neq res$. Let γ'' be an extension of $\alpha\gamma'$ such that p_l applies a sequence of read operations until one of these read operations returns a value $v' \neq \perp$, and then it applies the operation $op = CAS(v', w)$. By non-triviality the sequence of read operations in γ'' is not infinite and the operation $CAS(v', w)$ is successful.

Any CAS operation *op*" pending in $\alpha\gamma'$ or initiated in λ is of the form $CAS(v, -)$ with $v \neq w$. So, none of the CAS is linearizable. Then, *op* is the last successful CAS of $\alpha\gamma'\gamma''\lambda$.

Let res' be the value returned by the first response which is not $\perp$ when p_n performs a sequence of read operations solo after $\alpha\gamma'\gamma''\lambda$. Because of non-triviality this value exists. We have $res' = w$. By definition $res' \neq res$.

□

In the following we prove that any non-trivial solo-fast implementation of a deterministic abortable object that is perturbable has space complexity in $\Omega(n)$. The proof is similar to the proof of Theorem 4 in [3]. This proof does not directly apply because we consider a notion of solo-fast which is weaker than the one assumed in [3]. In particular, we authorize processes to use strong synchronization primitives even in absence of step contention if there is interval contention. But to avoid trivial solutions, a failed operation can enable only a finite number of other operations to be executed using strong synchronization primitives.

The following definition is needed for our proof. We say that a primitive is *writing* if its application may change the state of the object.

Definition 2. *A base object o is covered after an execution α if all the primitives applied to o in α are historyless, and there is a process p_n that has, after α, an enabled step e about to apply a historyless writing primitive to o. We also say that e covers o after α.*
An execution α is k-covering if there exists a set of processes $p_{j_1}, ..., p_{j_k}$ that does not contain process p_n, such that all the steps of α are applied by processes in this set and each of the processes in the set has an enabled writing step that covers a distinct base object after α.

Theorem 1. *Let A be an n-process non-trivial solo-fast implementation of a perturbable deterministic abortable object. The space complexity of A is at least $n - 1$.*

Proof. We prove the theorem by showing that A has an $(n-1)$-covering execution.

The proof goes by induction. The empty execution is vacuously a 0-covering execution. Assume that α_i , for $i < n - 1$, is an i-covering execution with covering set $\{p_{j_1}, ..., p_{j_i}\}$. Let λ_i be the execution fragment that consists of the writing steps by processes $p_{j_1}...p_{j_i}$ that are enabled after α_i , arranged in some arbitrary order.

Let $p_{j_{i+1}} \notin \{p_n, p_{j_1}, ..., p_{j_i}\}$ be a process that executes some steps solo after α_i. Because of the non-triviality of the solo-fast property after a finite number of steps the process $p_{j_{i+1}}$ uses only historyless primitives. We define $\delta_{j_{i+1}}$ the sequence of steps executed by $p_{j_{i+1}}$ after α_i after which $p_{j_{i+1}}$ has finished an operation (no enabled step) and it applies only historyless primitives (if any). In the same way, we define $\delta_{j_{i+x}}$ for $p_{j_{i+x}} \notin \{p_n, p_{j_1}, ..., p_{j_i}\}$ for $x = 2..n - i - 1$ relatively to $\alpha_i\delta_{j_{i+1}}...\delta_{j_{i+x-1}}$.

Let δ be the concatenation of all $\delta_{j_{i+x}}$ for $x = 1..n - i - 1$. We have ${\alpha_i}' = \alpha_i\delta$. After ${\alpha_i}'$ the process $p_{j_{i+x}}$ for $x = [2, n - i - 1]$ will apply historyless primitive.

By Definition 1, there is an execution fragment γ by some process $p_{j_x} \notin \{p_n, p_{j_1}, ..., p_{j_i}\}$ such that the first response different than abort, i.e. $\perp$, returned to p_n when repeatedly executing op_n respectively after $\alpha'_i\lambda_i$ and $\alpha'_i\gamma\lambda_i$ is different. We claim that γ contains a writing step that accesses a base object not covered after α'_i . We assume otherwise to obtain a contradiction. Since all steps in executions fragment λ_i and γ apply primitives from a historyless set, every writing primitive applied to a base object in γ is overwritten by some event in λ_i. Thus, the values of all base objects are the same after $\alpha'_i\lambda_i$ and after $\alpha'_i\gamma\lambda_i$. This implies that op_n must return the same response after both $\alpha'_i\lambda_i$ and $\alpha'_i\gamma\lambda_i$, which is a contradiction.

γ' is the shortest prefix of γ at the end of which p_{j_x} has an enabled writing step about to access an object o not covered after α'_i. We define α_{i+1} to be $\alpha'_i\gamma'$. Thus, at the end of α_{i+1}, p_{j_x} has an enabled writing step that accesses o. α_{i+1} is an execution, after which processes $p_{j_1}, ..., p_{j_i}, p_{j_x}$ have enabled steps that cover distinct objects. Hence, α_{i+1} is an $(i+1)$-covering execution. It follows that A has an $(n-1)$-covering execution. □

4 A Non-trivial Solo-fast Universal Construction (NSUC)

The algorithm uses read/write registers and CAS objects. A register R stores a value from some set and supports a read primitive which returns the value of R, and a write primitive which writes the value v in R. A CAS object support two primitives : $CAS(o, e, v)$ and $Read(o)$, where o is a CAS register,

e an expected value, and v a new value. If the value currently stored at a matches the expected value e, then $CAS(o, e, v)$ stores the new value v at o and returns $true$ (the CAS succeeds). Otherwise, CAS returns $false$ and does not modify the object (the CAS fails); $Read(o)$ returns the value stored in the CAS object and it does not modify the state of o.

The shared variables present in the algorithm are following :

- An array A of n single writer multireader (SWMR) registers. Each register contains a sequence value. In particular, process p_i announces its intention to change the current state of the shared object, by writing into location $A[i]$ the sequence that will be associated to the new state if p_i succeeds its operation. Our algorithm guarantees that each state of the object is univocally associated with a sequence. Initially, $A[j] = 0$ for $j = 1..n$.

- An array F of n SWMR registers. Each register contains a sequence and the pointer to a state of the shared object. The process p_i writes $< seq, \sigma >$ in $F[i]$ if it has detected that it is the first process to announce its intention to define the state for the sequence seq, σ is the proposed state. Initially, $F[j] =< 0, \perp >$ for $j = 1..n$.

- An array OS of n SWMR registers. If there is no contention process p_i writes $< seq, state >$ into $OS[i]$ where $state$ is the pointer to the new state of the shared object computed by p_i while executing its operation and seq is the associated sequence value. Initially, $OS[j] =< 0, \perp >$ for $j = 1..n$.

- A single CAS register OC which contains a sequence, an identifier of a process and a pointer to a state of the shared object. It is used in case of contention to decide the new state of the object among the ones proposed by the concurrent operations.

  In particular, if a process p_i detects the contention, it tries to change the state of the CAS register into a tuple ($< seq, id, newState >$) where id is the identifier of the process that proposes the state $newState$ associated to the sequence seq.

  id may be different than i if process p_i detects that another process p_{id} is concurrently executing an operation and both are trying to propose a new state for the same sequence value. p_i then helps the other process to apply its changes. Initially, $OC =< 0, 0, \sigma >$ where σ is the pointer to the initial state of the shared object.

- An array S of n CAS registers. Before trying to apply its changes to the CAS register OC, a process stores the sequence value stored in OC in S. Precisely, if the value of OC is $< seq, i, state >$, $S[i]$ will be set to seq. This is necessary to ensure that a process whose operation completes thanks to another process is aware that its operation succeeded. Thus, if $S[i] = seq$ process i knows that its operation which computed the state associated to seq succeeded.

  Initially, $S[j] = 0$ for $j = 1..n$.

We suppose to know if an operation op may changed the state of the shared object, or if it cannot . In the last case we say that op is *read-only*. This information is specified in the inputs.

```
   Code for process p_i to apply operation op with input arg on a DA object :
 1 < seq, state >← STATE() ;        //Find the last state and the sequence corresponding
 2 < newState, res >← APPLY_T(state, op, arg);
 3 if op is read-only then
 4 |  return res
 5 end
 6 seq ← seq + 1 ;                                                    //New sequence
 7 A[i] ← seq ;                                  //The process announces its intention
 8 id_new ← i;
 9 seq_A ← LEVEL_A(i);
10 if seq_A > seq then                    //A state is already decided for this sequence
11 |  return ⊥
12 end
13 if seq_A = seq then                                  //There is an interval contention
14 |  < id_F, newState_F >←WHOS_FIRST(seq);
15 |  if newState_F ≠ ⊥ then                              //Presence of a first process
16 |  |  newState ← newState_F;
17 |  |  id_new ← id_F;
18 |  end
19 else
20 |  F[i] ←< seq, newState >;
21 |  if LEVEL_A(i) < seq then                             //The process is still alone
22 |  |  OS[i] ←< seq, newState >;
23 |  |  return res
24 |  end
25 end
26 < seq_OC, id_OC, state_OC >←READ(OC);
27 while seq_OC < seq do
28 |  OLD_WIN(seq_OC, id_OC);
29 |  CAS(OC, < seq_OC, id_OC, state_OC >, < seq, id_new, newState >);
30 |  < seq_OC, id_OC, state_OC >←READ(OC);
31 end
32 if (seq_OC = seq ∧ id_OC ≠ i) ∨ (seq_OC > seq ∧ READ(S[i]) ≠ seq) then
33 |  res ← ⊥;
34 end
35 return res
```

Algorithm 1: NSUC - Code for process p_i

Description

At any configuration the state of the object is the value with the highest sequence stored either in the CAS register OC or in the array OS.

When a process p_i wants to execute an operation op on an object of type T, it first reads the current state of the object and the corresponding sequence seq (line 1). Then, p_i locally applies op to the read state (line 2). The algorithm assumes a function $APPLY_T(s, op, arg)$ that returns the response matching the invocation of the operation op in a sequential execution of op with input arg from state s for the object of type T. $APPLY_T(s, op, arg)$ also returns the new state of the object.

If the operation op cannot change the state of the object, p_i immediately returns the response (line 3 to 5). Otherwise, after incrementing the sequence value seq (line 6), p_i announces its intention to modify the state of the object by writing value $seq + 1$ into the register $A[i]$ (line 7). In particular, according to p_i's knowledge the current state of the object is the one associated to the sequence value seq and it is going to compete to decide the new state. This latter will be associated to the sequence value $seq + 1$.

After announcing its intention to modify the object, p_i checks if some other process is concurrently executing a non trivial operation on it. This is done by reading the other entries of the array A and looking for sequences greater than or equal to $seq + 1$.

Then, three cases can be distinguished.

- A greater sequence is found. Then some other process already decided the state for $seq + 1$ and p_i aborts.
- p_i detects that it is the first process announcing a proposal for the sequence $seq + 1$ (no read sequence is greater than or equal to p_i's one, i.e. $LEVEL_A(i) < seq + 1$). So p_i writes its proposal, $(seq + 1, newstate)$ into the register $F[i]$ (line 20) and checks again for concurrent operations (line 21). If p_i is still the only process to announce a proposal for $seq + 1$, it writes its proposal into the register $OS[i]$ (lines 21-22). This means that the state of the object associated to $seq + 1$ is the one proposed by p_i. This is because any other process competing for the same sequence will read that p_i is the first process to propose a new state for $seq+1$ and will help p_i to complete its operation (lines 15-17 and line 19). Finally, p_i returns the response of the operation (line 23).
- p_i reads $seq + 1$ in one of the other entries. Then it detects that another process is concurrently trying to decide the state for this sequence. If the detection is done on line 13, then p_i checks the presence of a process p_j competing for the sequence $seq+1$ and which has seen no contention (i.e. p_j has written its proposal in $F[j]$) in line 14. If this process exists, p_i will help p_j to apply its changes to the state of the object (lines 15 to 18). In particular, since there is contention p_i will try to write p_j's proposal into the CAS register OC (lines 26 to 31). Then it will return abort (lines 32 to 35). Otherwise p_i continues to compete for its own proposal. It tries to write the proposed state into OC (lines 26 to 31) until a decision is taken for the sequence $seq+1$. If a process (p_i or a helper) succeeds to perform a CAS in OC with p_i's proposal then p_i returns the response of its own operation (line 35). Otherwise it aborts. We have a similar behavior if a process detects the contention on line 21.

$STATE$ returns the current state of the shared object and its sequence.

```
   code for the function STATE()
1  seq_max ← 0;
2  σ_max ← ⊥;
3  for j = 1..n do
4  |  < seq_OS, σ > ← OS[j];
5  |  if seq_OS > seq_max then seq_max ← seq_OS; σ_max ← σ; end
6  end
7  < seq_OC, id_OC, σ_OC > ← READ(OC);
8  if seq_OC < seq_max then return < seq_max, σ_max > end
9  return < seq_OC, σ_OC >
```

Algorithm 2: function $STATE$

$LEVEL_A(i)$ returns the highest sequence written into the announce array A by a process other than p_i when.

```
   code for the function LEVEL_A(i)
1  seq_max ← 0;
2  for j = 1..n | j ≠ i do
3  |  seq_A ← A[j];
4  |  if seq_max < seq_A then seq_max ← seq_A; end
5  end
6  return seq_max
```

Algorithm 3: function $LEVEL_A$

For a given sequence seq $WHOS\_FIRST(seq)$ returns the couple (j, σ) where j is the first process (if any) to propose a new state for seq and σ is the proposed stated. For any given sequence value, the algorithm ensures that at most one such process exists. This is proved in Lemma 4.

```
   code for the function WHOS_FIRST(seq)
1  for j = 1..n do
2  |  < seq_F, σ_F > ← F[j];
3  |  if seq = seq_F then return < j, σ_F > end
4  end
5  return < 0, ⊥ >
```

Algorithm 4: function $WHOS\_FIRST$

OLD_WIN tries to write seq_{OC} in the CAS $S[id_{OC}]$ if $S[id_{OC}]$'s value is smaller than seq_{OC}. This ensures that a slow process p whose operation succeeded to modify the CAS OC and then the state of the implemented object is aware that its operation was successfully executed. In fact, it may happen that p did not take steps while another process completed its operation and, then another operation overwrote its changes by writing into OC. Then, p can recover the state of its operation checking into its location in S and return the correct response.

```
   code for the function OLD_WIN(seq_OC, id_OC)
1  seq_S ← READ(S[id_OC]);
2  if seq_OC > seq_S then CAS(S[id_OC], seq_S, seq_OC); end
```

Algorithm 5: Function OLD_WIN

Complexity

Let t be the worst case time complexity to perform an operation on the sequential implementation of the object (i.e. the time complexity of the function $APPLY_T$). Because of the n iterations in the function $STATE$, $LEVEL_A$ and $WHOS\_FIRST$, the time complexity of these functions is $O(n)$. On the contrary the time complexity for the function OLD_WIN is $O(1)$, then during each iteration of the loop (lines 27 to 31 of the Algorithm 1) the time complexity is $O(1)$. Since a process can repeat the loop at most n times (according to Lemma 2 and Lemma 3 in the appendix), the worst time complexity for the loop is $O(n)$. And so, the time complexity is $O(n+t)$. Since for any operation, a process executes at least $STATE$ and $APPLY_T$, the time complexity is $\Omega(n+t)$. Then, we have a time complexity $\Theta(n+t)$.

Let s be the size of the sequential representation of the object. The NSUC algorithm stores at most $2n+1$ sequential representation of the object (n for the array F, n for the array OS and 1 for OC). So the space complexity of NSUC algorithm is in $O(ns)$.

5 Proofs of NSUC

In the following all the line numbers refer to Algorithm 1 unless specified.

5.1 Wait-freedom

For any pair of tuples t_1, t_2 stored in OC or in OS we say that t_1 is greater than t_2 iff the sequence in t_1 is greater than the sequence in t_2.

Observation 1 *The values written in the CAS object OC are increasing.*

This follows from the fact that the CAS object OC is written only at line 29 and because of line 27 the value written is greater than the value stored in OC immediatly before the write succeeded.

Observation 2 *$\forall i = 1..n$ the values written in $OS[i]$ are increasing.*

$OS[i]$ is written only by process p_i. Inspection of the pseudocode of the function $STATE$ reveals that a process p_i begins the execution of an operation *op* with a value of *seq* equal to or greater than the value in $OS[i]$ at the configuration before the invocation of *op*. Before writing a new sequence value in $OS[i]$ the only operation on *seq* is an increment (line 6 of Algorithm 1).

Lemma 2. *A process p stays in the loop (lines 27 to 31) only if another process q succeeds the CAS at line 29 with a sequence value smaller than the seq value of p when executing line 27, in between the last read of OC by p and the last CAS operation to OC by p.*

Proof. Let us assume that no process succeeds a CAS on OC between the last read of OC by p and the application of CAS on OC by p. Then process p succeeds its CAS and writes a tuple with sequence *seq* into OC. p exits the loop because by Observation 1 the value it successively reads in OC is greater than or equal to *seq*. Similarly, if a process succeeds the CAS with a sequence greater or equal than *seq*, then by Observation 1, p exits the loop. □

Lemma 3. *Let p be a process in the loop (lines 27 to 31). Another process q can prevent p to exit the loop at most once.*

Proof. According to Lemma 2, a process q can prevent p to exit the loop only if it succeeds its CAS with a sequence smaller than the value of *seq* of p, in between the time of the last read of OC by p and the following CAS by p. Let v be the value of the *seq* of p when p executes line 27. Assume that a process q writes into OC a sequence value smaller than v in between the last read of OC by p and its successive application of CAS to OC. After its CAS, q exits the loop by Observation 1 and its operation is terminated. If q executes a new operation, it will obtain (at line 1) a sequence value greater than or equal to v. This is because of Observation 1, Observation 2 and by the pseudocode of the function $STATE$. Then q cannot prevent anymore p to exit the loop. □

Lemma 2 and Lemma 3 prove the following theorem.

Theorem 2. *Every invocation of an operation by a non-faulty process returns after a finite number of its own steps.*

5.2 Deterministic Abortable Object

In this section, we prove that every operation that aborts (i.e. returns $\perp$) does not modify the state of the implemented object.

Lemma 4. *For any given integer value v at most one process writes v into F.*

Proof. A process p_i writes v into F (line 20 of Algorithm 1) only if the seq_A value read line 9 is smaller than v. Observe that p_i writes v in $A[i]$ (line 7) before executing line 9 of Algorithm 1.

Assume by contradiction that there is another process p_j that writes v into F. Then, since both p_i and p_j verify the condition $seq_A < v$, none of them read the value v written into A by the other process. This means that the read of $A[i]$ ($A[j]$) by p_j (p_i) precedes the write of $A[i]$ ($A[j]$) by p_i (p_j). Since the write of $A[i]$ precedes the read of $A[j]$ by p_i and similary from p_j, we reach a contradiction. □

Lemma 5. *Let t, t' be two tuples written into OC and/or into OS. If t and t' have a same sequence value then the state field of t is equal to the state field of t'.*

Proof. By Observation 1 for any given integer value v at most one tuple with sequence value v is written into OC. By Lemma 4 the same holds for the tuples written into OS. It remains to prove that for any pair of tuples t and t' with a same sequence value and such that t is written into OC and t' is written into OS, the state field of t and t' is the same.

If there is a state s associated to a sequence value v which is written in OC by process p_i and a state s' associated to a sequence value v which is written in OS by process p_j. Then p_j has previously written $< v, s' >$ into $F[j]$ (line 20) and before this write p_j wrote v into A (line 7). Since at line 21 p_j reads a sequence value smaller than v, then the read of $A[i]$ precedes the write of v into $A[i]$ by p_i. Since p_i first writes v into $A[i]$ and then reads $A[j]$, the value read in $A[j]$ is greater than or equal to v. Since p_i writes into OC (i.e. it executes line 29), the value read in $A[j]$ is equal to v. Then, it executes lines 13-18 and by Lemma 4 and according to the pseudocode of the function $WHOS\_FIRST$, we have that at line 14 p_i reads the state s written by p_j. Thus $s = s'$. □

Observation 3 *A sequence value v cannot be written in $S[i]$ if no process has written in OC the triplet $< v, i, state >$.*

This follows from lines 26, 28 of Algorithm 1 and line 3 of Algorithm 5.

Observation 4 *$\forall\ i = 1..n$ the values written in $S[i]$ are increasing.*

This follows from lines 2,3 of Algorithm 5.

Lemma 6. *Consider an invocation to $OLD\_WIN(v, i)$ by a process p which returns. If p, while executing $OLD\_WIN(v, i)$ reads $S[i] < v$ on line 1, then either p succeeds the $CAS(S[i], -, v)$ on line 3 or some other process wrote v into $S[i]$ at a configuration that precedes the application of this CAS.*

Proof. Let $s_1, \dots s_h$ be the sequence of tuples with process identifier i written into OC ordered according to their sequence number. By Observation 1 this order corresponds to the order these tuples are written into OC. Thus, s_k with $k = 1, \dots h$ is the k-th tuple with identifier i written into OC. The proof is by induction on k.

In the *base case* $k = 1$. Let C be the first configuration at which the value of OC is s_1. Let q be the process that overwrites the value s_1 in OC. Before applying this write operation on OC, q invokes $OLD\_WIN(s_1.seq, i)$ where $s_1.seq$ is the sequence number of s_1. Since q is the process that overwrites s_1 and by Observation 3, the value read by q in $S[i]$ is the initial value and then it is smaller than $s_1.seq$. Suppose q fails the $CAS(S[i], -, s_1.seq)$. Then, by Observation 3 another process has written $s_1.seq$ into $S[i]$ before q applies its CAS. Thus, when q returns from $OLD\_WIN(s_1.seq, i)$ the value $s_1.seq$ has been written into $S[i]$.

Consider any other process p that invokes $OLD\_WIN(s_1.seq, i)$ and returns from this call. Assume p reads a value smaller than $s_1.seq$ at line 1 and it fails the $CAS(S[i], -, s_1.seq)$. If it fails after q returns from $OLD\_WIN(s_1.seq, i)$, the claim follows. Otherwise, by Observation 3 another process has written $s_1.seq$ into $S[i]$ in between its read operation and its application of the CAS.

For the *induction step*, Let p be a process that executes $OLD\_WIN(s_k.seq, i)$ and returns from this invocation. Assume that p reads a value $S[i] < s_k.seq$ and fails the $CAS(S[i], -, s_k.seq)$. To invoke $OLD\_WIN(s_k.seq, i)$ p has previously read s_k into OC (line 26 of Algorithm 1). Then, by Observation 1, the tuple s_{k-1} was written into OC before this read operation.

Let C be the first configuration at which the value of OC is s_{k-1}. The process that overwrites this tuple has to previously invoke and return from the call of $OLD\_WIN(s_{k-1}.seq, i)$. Thus, by the inductive hypothesis and Observation 3 at some configuration before the write into OC of the value s_k, the value of $S[i]$ is equal to s_{k-1}.

If p is the process that overwrites the value s_k in OC. Then, by Observations 3 and 4 another process has written $s_k.seq$ into $S[i]$ before p applies its CAS. Otherwise p is not the process that overwrites the value s_k into OC. If it fails the CAS after s_k is written into OC, then the claim follows. Otherwise, by Observations 3 and 4 another process has written $s_k.seq$ into $S[i]$ before p applies its CAS. □

When a process p_j succeeds a CAS in OC overwritting $< v, i, - >$, $S[i]$ contains v.

Lemma 7. *Let C be the configuration immediately after a successful application of $CAS(OC, < v, i, - >, -)$. Then $S[i] = v$ at C.*

Proof. By Observation 1 a tuple $< v, -, - >$ can be written into OC only once. Let C be the configuration immediatley after the successful application of $CAS(OC, < v, i, - >, < -, -, - >)$ by a process p. Since the CAS is successful there is a configuration C' that precedes C such that the value of the CAS object OC is $< v, i, - >$ at C' and in between C' and C no CAS operation on OC succeed.

Inspecting the pseudocode of Algorithm 1 reveals that p executes $OLD\_WIN(v, i)$ before C. By Observations 1 and 4 the value read by p in $S[i]$ (line 1 of Algorithm 5) is less than or equal to v. If the value read by p is less than v, then by Lemma 6 either p succeeds the CAS at line 3 of Algorithm 5 writing v into $S[i]$ or another process did it at some configuration preceding the application of the CAS on $S[i]$ by p. Since no other CAS operation succeeds in between C' and C and because of Observation 4, the value of $S[i]$ is v at C. □

Let p_i be a process executing an operation. We define seq^i_{op} as the sequence value that the function $STATE$ returns on line 1 during the execution of an operation instance op by process p_i. If there is no ambiguity or we are not interested in the process executing op we use seq_{op}.

We denote $seq_O(C)$ the greatest sequence value stored in the CAS register OC and in the array OS at configuration C and $seq_A(C)$ the greatest value in A at configuration C.

Lemma 8. *For all positive integer v if $seq_O(C) \geq v$, then a process has written $< v, -, - >$ into OC or $< v, - >$ into OS before configuration C.*

Proof. We do the proof by backward induction on v. If $v = seq_O(C)$ the claim is true by definition of $seq_O(C)$. We assume that for a v such that $v \leq seq_O(C)$ the claim is true and we prove that it is true for $v - 1$. Let p be the process which has written $< v, -, - >$ into OC or $< v, - >$ into OS before the

configuration C. Before this write operation p has read the value $v-1$ in OC or in OS when executing line 1 of Algorithm 1. Inspecting the pseudocode of the function $STATE$ reveals that before this read operation and then before configuration C a process wrote $< v-1, -, - >$ into OC or $< v-1, - >$ into OS □

The following theorem states that an operation instance that aborts does not change the state of the shared object.

Theorem 3. *If an operation instance op executed by a process p aborts, then the tuple with the new state computed by p while executing op will never be written into OC or into OS.*

Proof. A process can abort on line 11 or line 33. If the process aborts on line 11, it is trivial.

If the process p_i aborts on line 33 either $(v_{OC} = seq^i_{op} + 1 \wedge id_{OC} \neq i)$ or $(v_{OC} > seq^i_{op} + 1 \wedge READ(S[i]) \neq seq^i_{op} + 1)$, where v_{OC} and id_{OC} are respectively the sequence value and the identifier of the last tuple p_i read in OC.

- In the first case, p_i has read the state associated to its sequence value $seq^i_{op} + 1$ and its not its state. By Lemma 5 its proposed state will never be written in OC or in OS.
- In the second case, the sequence value read by p_i in OC is greater than $seq^i_{op} + 1$ and $S[i]$ does not contain $seq^i_{op} + 1$. Then according to Lemma 7 and Observation 4, no process has written into OC the state proposed by p_i for the sequence value $seq^i_{op} + 1$. According to Lemma 8, a tuple with sequence value equal to $seq^i_{op} + 1$ has been written into OS or into OC. Therefore the state proposed by p_i for the sequence number $seq^i_{op} + 1$ will never be written into OC or into OS (Lemma 5).

□

5.3 Non-triviality

In this section we prove that our algorithm is non-trivial according to the definition of non-triviality proposed in [2]. In particular, an operation can abort only if it is concurrent with another operation, and an operation that does not complete can cause only a finite number of operations to abort. In particular, we prove that it can cause the abort of at most two operations per process.

Observation 5 *At any configuration C, we have* $seq_A(C) \geq seq_O(C)$.

This follows from the fact that every process writes the greatest value read in OC and in OS plus one in A before writing into OC or into OS.

Lemma 9. *For any given execution* α*, and any configuration* C *in* α *we have* $seq_A(C) = seq_O(C)$ *or* $seq_A(C) = seq_O(C) + 1$

Proof. Fix an execution. Initially this is true : $seq_A(0) = 0$ and $seq_O(0) = 0$. Assume that is true up to configuration C. Two cases have to be studied.

- The step that brings from configuration C to configuration $C+1$ is a write of a sequence value v into OC or into OS by a process p.
  Then $seq_A(C) = seq_A(C+1)$. Also, according to Observation 1 and Observation 2, we have $seq_O(C+1) \geq seq_O(C)$. If $seq_O(C+1) = seq_O(C)$, then the claim is trivially true. Otherwise $seq_O(C+1) > seq_O(C)$, and then $seq_O(C+1) \geq seq_A(C+1)$. By Observation 5 $seq_A(C+1) \geq seq_O(C+1)$. Then $seq_A(C+1) = seq_O(C+1)$.
- The step that brings from configuration C to configuration $C+1$ is a write of a sequence value v into A by a process p_i while executing an operation op. This implies that $seq_O(C) = seq_O(C+1)$. Also, $v = seq^i_{op} + 1$ and $seq_O(C) \geq seq^i_{op}$ because seq^i_{op} has been read before configuration C.
  - First consider $seq_O(C) = seq^i_{op}$. If the value written by p_i is not greater than the values stored in A at C, then $seq_A(C) = seq_A(C+1)$. Otherwise, we have $seq_A(C+1) = seq^i_{op}+1 = seq_O(C+1)+1$. In both cases the claim follows.

- If $seq_O(C) > seq^i_{op}$, then there is a process p_j which has written $seq_O(C)$ in OC or in OS before the configuration C and after the configuration that immediately follows the read by p_i on line 1. Then p_j has written $seq_O(C)$ in $A[j]$ before the configuration C. Also p_j is not the process p_i otherwise $seq_{op} = seq_O(C)$ and we reach a contradiction. Given that $i \neq j$, and since the value written by p_i is smaller than or equal to $seq_O(C)$, we have $seq_A(C) = seq_A(C+1)$. The claim follows.

□

The following Lemma states that when an operation that tries to write the *ith* state of the object aborts, the *ith* state is already defined. Thus, our algorithm ensures that even in presence of interval contention at least one operation succeeds to modify the state of the object.

Lemma 10. *Let op be an operation that may change the state of the object executed by a process p. Let C be the configuration immediately after op returns. A tuple corresponding to the sequence value* $seq_{op}+1$ *has been written into OC or into OS before C.*

Proof. Let p be a process that executes an operation *op* that may change the state of the object . *op* returns because p executes one of the following lines : line 11, line 23 or line 35 of Algorithm 1. Let C be the configuration immediately before *op* returns.

- If *op* returns on line 23 : Before returning, process p has written into its entry of OS a tuple $< seq_{op}+1, ->$ (line 22).
- If *op* returns on line 11 : The value read by process p at line 9 is greater than $seq_{op}+1$. According to Lemma 9, we have $seq_A(C) = seq_O(C)$ or $seq_A(C) = seq_O(C)+1$, consequently we have $seq_O(C) \geq seq_{op}+1$. By Lemma 8, a process has written a tuple with the sequence value $seq_{op}+1$ into the CAS register OC or into the array OS before configuration C.
- *op* returns on line 35 : Because of line 27 of Algorithm 1, we have $seq_O(C) \geq seq_{op}+1$. Then, by Lemma 8, a process has written a tuple with the sequence value $seq_{op}+1$ in the CAS register OC or into the array OS before configuration C.

□

Lemma 11. *For any given execution* α*, let op be an operation instance executed by process* p_i*. If the value of* $LEVEL_A(i)$ *reads by* p_i *at some point during the execution of op is greater than or equal to* $seq^i_{op}+1$*, then op is concurrent with another operation* op' *in* α*.*

Proof. Let p_i be a process that executes an operation *op*. Assume that p_i reads $LEVEL_A(i) = v_A \geq seq^i_{op}+1$ during the execution of *op*. Then, another process p_j has written the value v_A into $A[j]$ while executing an operation instance op'.

Assume by contradiction that *op* and op' operations are not concurrent. Since we assume they are not concurrent and as the process p_i reads a value written by p_j during op', op' precedes *op*. Inspecting the code reveals that *op* and op' are not trivial operations since they access the array A. According to the Lemma 10, when op' terminates at configuration C, $seq_O(C) = v_A$. So, at configuration C' where p_i starts its operation $seq_O(C') \geq v_A$ and seq^i_{op} should be greater than or equal to v_A. This is a contradiction. □

Observation 6 *An operation instance op executed by a process* p_i *aborts only if* p_i *reads* $LEVEL_A(i) \geq seq^i_{op}+1$ *at some point during the execution of op.*

This follows from the fact that if an operation instance aborts on line 11, the process has executed line 10; if an operation instance aborts line 33 the process executed line 13 or line 21.

Lemma 12. *For any given execution* α*, let op be an operation instance by process p that aborts in* α*. Then op is concurrent with some other operation* op' *in* α*.*

Proof. Let *op* be an operation instance by process p_i that aborts. By Observation 6 p_i reads at some point during the execution of *op* $LEVEL_A(i) \geq seq^i_{op}+1$. Moreover by Lemma 11 *op* is concurrent with another operation op' in α. □

Lemma 13. *Fix an execution α. Let p_i be a process that fails in α while executing an operation op. For any non faulty process p_j let op_1 be the first operation executed by p_j which is concurrent with op. Let op_2 and op_3 be two consecutive operation instances executed by p_j immediately after op_1. We have that $seq^j_{op_3} > seq^i_{op}$.*

Proof. Let p_i be a process that fails in α while executing an operation *op*. By Theorem 2 every operation executed by a non-faulty process terminates. Let p_j be a non-faulty process in α. Observe that the invocation of op_2 by p_j follows the invocation of *op* by p_i. Thus, by Observations 1 and 2 $seq^j_{op_2} \geq seq^i_{op}$. Assume that $seq^j_{op_2} = seq^i_{op}$ (otherwise the claim is proved). Let C' be the configuration immediately after op_2 returns, by Lemma 10 $seq_O(C') > seq^j_{op_2}$. Then, inspecting the code of the function $STATE$ reveals that $seq^j_{op_3} > seq^i_{op}$. □

Lemma 14. *Fix an execution α. Let p_i be a process that fails in α while executing an operation instance op. Then for any non faulty process p_j, op can abort at most two operations of p_j.*

Proof. Let p_i be a process that fails and p_j an other process. If $seq^i_{op} < seq^j_{op}$, the process p_i cannot cause the process p_j to abort. This follows from Observation 6 and the fact that $seq^i_{op} < seq^j_{op}$. In fact, since $A[i] \leq seq^i_{op} + 1 < seq^j_{op} + 1$, p_i cannot cause the read by p_j of $LEVEL_A(j) \geq seq^j_{op} + 1$. The claim follows from Lemma 13. □

Lemma 12 and Lemma 14 prove the following theorem.

Theorem 4. *The Algorithm NSUC is non-trivial.*

5.4 Non-trivial solo-fast

In this section, we prove that our algorithm is non-trivial solo-fast. Informally, this means that during the execution of an operation a process applies some no-histoyless primitives only if this operation is concurrent with another one. Moreover, an operation *op* that does not complete can cause a process to apply no-histoyless primitives for at most two consecutive operations concurrent with *op*.

Observation 7 *A process p_i applies no historyless primitive while executing an operation op only if it reads $LEVEL_A(i) \geq seq^i_{op} + 1$ at some point during the execution of op.*

This follows from the fact that a process applies no historyless primitive only on lines 28 and 29. Then, to execute these lines it must have executed line 13 or line 21.

The following Lemma holds by Observation 7 and Lemma 11, in the same way we have proved Lemma 12 :

Lemma 15. *For any given execution α, let p be a process that applies no historyless primitive while executing an operation instance op. Then op is concurrent with some other operation op' in α.*

The following Lemma holds by Observation 7 and Lemma 13, in the same way we have proved 14 :

Lemma 16. *Fix an execution α. Let p_i be a process that fails in α while executing an operation instance op. For any process p_j, let op_1,op_2 and op_3 be three consecutive operations executed by p_j concurrently with op. p_j applies no historyless primitives when executing op_3 only if it exists an operation $op' \neq op$ such that the invocation of op' follows the response of op_1 and op' is concurrent with op_3.*

Informally, this Lemma states that an operation *op* that does not complete can justify the application of no historyless primitives by a process for the execution of at most two operations concurrent with *op*.

Lemma 15 and Lemma 16 prove the following theorem.

Theorem 5. *The Algorithm NSUC is non-trivial solo-fast.*

5.5 Linearizability

Fix an execution α, for any configuration $C \in \alpha$, by Lemma 8 and Lemma 5, $\forall 0 < v \leq seq_O(C)$ there is a unique state *state* such that a process has written $< v, -, state >$ into OC or $< v, state >$ into OS before configuration C. We say that the operation that change the state of the object to the state *state* such that a proces has written $< v, -, state >$ into OC or $< v, state >$ into OS, is associated to the sequence v and also that the state *state* is associated to v.

Let π be a permutation of the high-level operations in α. We construct π by distinguishing three type of operation : the read-only operations, the operations that are not read-only and return $res \neq \perp$ and the operations that are not read-only and return $\perp$.

First, the operations that are not read-only and return $res \neq \perp$ change the state of the object, so they are associated to a sequence value. We order these operations according to the ascending order on the sequence value associated to them.

Secondly, we consider each read-only operation in the order in which its reponse occurs in α. A read-only operation *op* is placed immediately before the operation associated to the sequence value $seq_{op} + 1$.

Finally, an operation *op* that is not read-only and returns $\perp$ is placed immediately before the operation associated to the sequence value $seq_{op} + 1$.

Observation 8 *Let op and op′ be two operations in α. By construction of π if $seq_{op} < seq_{op'}$ then op precedes op′ in π .*

Observation 9 *Fix an execution α, for any configuration $C \in \alpha$ and for any $v \leq seq_O(C)$, we have that by construction of π, the last operation op in π such that $seq_{op} = v$ is the operation associated to the sequence value $seq_{op} + 1$.*

Lemma 17. *Let s be the state of the object of type T^{da} before the operation op in π. Then the response res returns by op in α is such that $(s, op, s', res) \in \Delta$ where Δ is the sequential specification of T^{da}.*

Proof. Let Δ be the sequential specification of T^{da}. If *op* is an operation that is not read-only and return $\perp$, then by definition of type T^{da}, for any state s of the object there is $(s, op, s, \perp) \in \Delta$

If *op* is an operation that is not read-only and returns $res \neq \perp$, then it is associated to a sequence value v and $seq_{op} = v - 1$. Then res is such that $(s, op, s', res) \in \Delta$ with s the state associated to the sequence value $v-1$ or the initial state if $v = 1$. By observation 8 the previous state is the state associated to $v - 1$ or the inital state if $v = 1$. So, the claim is true.

If *op* is a read-only operation, then res is such that $(s, op, s, res) \in \Delta$ with s the state associated to the sequence value seq_{op} or the initial state if $seq_{op} = 0$. By Observations 8 and 9, the previous state is the state associated to seq_{op} or the inital state if $seq_{op} = 0$. So, the claim is true. □

Lemma 18. *Let op and op′ be two high-level operation in α. If the invocation of op′ follows the response of op in α, then op precedes op′ in π.*

Proof. Let the invocation of an operation *op′* follows the response of an operation *op* in α.

First assume that *op* is not read-only. Then by Lemma 10, Observations 1 and 2, we have $seq_{op} < seq_{op'}$. By observation 8, then *op* precedes *op′* in π .

Then assume that *op* is read-only. Then by Observations 1 and 2, we have $seq_{op} \leq seq_{op'}$. If $seq_{op} < seq_{op'}$, by observation 8, then *op* precedes *op′* in π . Let consider $seq_{op} = seq_{op'}$. Two cases can be distinguished. First, we consider that *op′* is read-only. The response of *op* occurs in α before the reponse of *op′*, then by construction of π for read-only operations, *op* precedes *op′*. Finally, if *op′* is not read-only, by construction of π, it is placed after all read-only operations with the same sequence value seq_{op}. We have also that *op* precedes *op′* in π. □

Lemma 17 and Lemma 18 prove the following theorem.

Theorem 6. *The Algorithm NSUC is linearizable.*

6 Conclusion

We have studied solo-fast implementations of deterministic abortable objects. We have investigated the possibility for those implementations to have a better space complexity than linear if relaxing the constraints for a process to use strong synchronization primitives.

We have proved that solo-fast implementations of some deterministic abortable objects have space complexity in $\Omega(n)$ even if we allow a process to use strong synchronization primitives in absence of step contention, provided that its operation is concurrent with another one. To prove our results we consider only non-trivial implementations, that is implementations where a crashed process can cause only a finite number of concurrent operations to abort.

Then, we have presented a non trivial solo-fast universal construction for deterministic abortable objects.

Any implementation resulting from our construction is *wait-free*, *non-trivial* and *non-trivial solo-fast* : without interval contention, an operation uses only read/write registers; and a failed process can cause the use of CAS and the abort of at most two operations per process. Moreover, our universal construction ensures that at least one writing operation succeeds to modify the object also in case of contention. And, any operation that does not change the state of the objet always returns a legal response and does not use strong synchronization primitives.

If t is the worst time complexity to perform an operation on the sequential object, then $\Theta(t + n)$ is the worst time compexity to perform an operation on the resulting object. If the sequential object has size s, then the resulting object implementation has space complexity in $O(ns)$. This is asymptotically optimal if the implemented object has constant size.

References

1. Afek, Y., Stupp, G., Touitou, D.: Long lived adaptive splitter and applications. Distributed Computing 15(2), 67–86 (2002), http://dx.doi.org/10.1007/s004460100060
2. Aguilera, M.K., Frolund, S., Hadzilacos, V., Horn, S.L., Toueg, S.: Abortable and query-abortable objects and their efficient implementation. In: the 26th ACM Symposium on Principles of Distributed Computing (PODC'07). pp. 23–32 (2007), http://doi.acm.org/10.1145/1281100.1281107
3. Attiya, H., Guerraoui, R., Hendler, D., Kuznetsov, P.: The complexity of obstruction-free implementations. J. ACM 56(4), 24:1–24:33 (2009), http://doi.acm.org/10.1145/1538902.1538908
4. Attiya, H., Guerraoui, R., Kouznetsov, P.: Computing with reads and writes in the absence of step contention. In: the 19th International Conference on Distributed Computing (DISC'05). pp. 122–136 (2005)
5. Chuong, P., Ellen, F., Ramachandran, V.: A universal construction for wait-free transaction friendly data structures. In: the 22nd ACM Symposium on Parallelism in Algorithms and Architectures (SPAA'10). pp. 335–344 (2010), http://doi.acm.org/10.1145/1810479.1810538
6. Crain, T., Imbs, D., Raynal, M.: Towards a universal construction for transaction-based multiprocess programs. Theor. Comput. Sci. 496, 154–169 (2013)
7. Fich, F., Herlihy, M., Shavit, N.: On the space complexity of randomized synchronization. J. ACM 45(5), 843–862 (1998), http://doi.acm.org/10.1145/290179.290183
8. Guerraoui, R., Kapalka, M.: The semantics of progress in lock-based transactional memory. In: the 36th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'09). pp. 404–415 (2009)
9. Hadzilacos, V., Toueg, S.: On deterministic abortable objects. In: the 2013 ACM Symposium on Principles of Distributed Computing (PODC'13). pp. 4–12 (2013), http://doi.acm.org/10.1145/2484239.2484241
10. Herlihy, M.: Wait-free synchronization. ACM Trans. Program. Lang. Syst. 13(1), 124–149 (1991), http://doi.acm.org/10.1145/114005.102808
11. Herlihy, M., Luchangco, V., Moir, M.: Obstruction-free synchronization: Double-ended queues as an example. In: the 23rd International Conference on Distributed Computing Systems (ICDCS'03). pp. 522–529 (2003)
12. Herlihy, M., Moss, J.E.B.: Transactional memory: Architectural support for lock-free data structures. In: the 20th Annual International Symposium on Computer Architecture (ISCA'93). pp. 289–300 (1993)
13. Herlihy, M., Wing, J.M.: Linearizability: A correctness condition for concurrent objects. ACM Transactions on Programming Languages and Systems 12(3), 463–492 (1990)
14. Jayanti, P., Tan, K., Toueg, S.: Time and space lower bounds for nonblocking implementations. SIAM J. Comput. 30(2), 438–456 (2000)
15. Luchangco, V., Moir, M., Shavit, N.: On the uncontended complexity of consensus. In: the 17th International Symposium on Distributed Computing (DISC03). pp. 45–59 (2003)
16. Perelman, D., Fan, R., Keidar, I.: On maintaining multiple versions in STM. In: the 29th ACM Symposium on Principles of Distributed Computing (PODC'10). pp. 16–25 (2010)