Solo-fast Universal Constructions for Deterministic Abortable Objects*

Claire Capdevielle, Colette Johnen, and Alessia Milani

Univ. Bordeaux, LaBRI, UMR 5800, F-33400 Talence, France
(firstname.lastname@labri.fr)

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 [3]. 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.

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.

* This work was partially supported by the ANR project DISPLEXITY (ANR-11-BS02-014). This study has been carried out in the frame of "the Investments for the future" Programme IdEx Bordeaux CPU (ANR-10-IDEX-03-02).

On the other hand, as pointed out by Attiya et al. in [3], 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 [3]. Thus, we study implementations that use only read/write registers when there is no contention and use stronger synchronization primitives, e.g., Compare and Swap (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 [3] 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.

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. We only consider 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].

Our results. 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 and we prove that a k-CAS abortable register is perturbable according to our definition.

Then, we present a solo-fast universal construction for deterministic abortable objects, called NSUC. 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. The NSUC 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 with 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 [3]. Few variants of these objects have been proposed [2, 3, 9]. The ones proposed in [2, 3] 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 can be easily transformed into solo-fast by using the solo-fast consensus object proposed in [3], but it 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 the solo-fast consensus object by Attiya at al, which has $\Theta(n)$ space complexity, we cannot easily modify existing universal constructions 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 un-

bounded space complexity. Finally, the NSUC algorithm ensures *multi-version permissiveness* 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 lower bound on the space complexity. In Section 4 we present our solo-fast universal construction. Finally, a sketch of the correctness proof of our construction is 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 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 base object o the set of its primitives 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 applying f' and then f results in the same state of o 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 primitive/operation is a *writing* primitive/operation if its application may change the state of the object. Otherwise it is a *read-only* primitive/operation.

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. A step e by

a process p is *enabled* at a given configuration C, if p is about to apply e at C. 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 $C_i, \phi_i, C_{i+1}, \phi_{i+1}, \ldots, \phi_{j-1}, C_j$ 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 configuration C, for any finite execution α ending with C and any execution α' starting at C, 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 constructions that guarantee 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 any given execution α every operation that aborts is concurrent with some other operation in α, and an operation that remains incomplete, due to a crash, 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 any given execution α a process p applies some non-historyless primitives while performing an instance of an operation *op*, only if *op* is concurrent with some other operation in α; and an operation that remains incomplete, due to a crash, does not justify the application of non-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 some process $p_l \neq p_n$ applies no step in λ and no process applies more than a single step in λ, there is an extension of α, γ, consisting of steps by p_l, such that the first response $res \neq \perp$, that p_n returns when repeatedly performing op_n solo after $\alpha\lambda$ is different from the first response $res' \neq \perp$ it returns when repeatedly performing op_n solo after $\alpha\gamma\lambda$.*

By adjusting the proof of Lemma 4.7 in [14], in [4] 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 $u \neq s\} \cup \{(s, Read, s, \perp)\} \cup \{(s, CAS(u, v), s, \perp)\}$.

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].

The following definition is needed for our proof.

Definition 2. *A base object o is covered after an execution α if the set of primitives applied to o in α is historyless, and there is a process that has, after α, an enabled step e about to apply a 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 $\{p_{j_1}, ..., p_{j_k}\}$, called, covering set*, such that each process in the covering 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. To prove the theorem we construct an execution which is p_n-free and $(n-1)$-covering. The proof goes by induction. The empty execution is vacuously a 0-covering execution and it is p_n-free. Assume that α_i, for $i < n - 1$, is an i-covering execution with covering set $\{p_{j_1}, ..., p_{j_i}\}$ and is p_n-free. 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 $p_{j_{i+1}}$ be a process not in $\{p_n, p_{j_1}, ..., p_{j_i}\}$. Since $i < n - 1$, this process exists. Because of the non-triviality of the solo-fast property process $p_{j_{i+1}}$ applies only historyless primitives after a finite number of its own steps when executing solo after α_i. Let δ be the shortest execution by $p_{j_{i+1}}$ when executing solo after α_i such that $p_{j_{i+1}}$ applies only historyless primitives (if any) after $\alpha_i\delta$ if still

running solo. $\alpha_i{}' = \alpha_i\delta$ is p_n-free and the writing steps of processes $p_{j_1}...p_{j_i}$ are enabled at the configuration immediately after $\alpha_i{}'$.

By Definition 1, there is an extension of $\alpha_i{}'$, γ, by $p_{j_{i+1}}$ such that the first response different than abort 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 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.

We denote γ' the shortest prefix of γ at the end of which $p_{j_{i+1}}$ has an enabled writing step about to access an object not covered after α_i'. We define α_{i+1} to be $\alpha_i'\gamma'$. α_{i+1} is a p_n-free execution and it is $(i+1)$-covering. This latter property is true because at the configuration immediately after α_{i+1} processes $\{p_{j_1}, ..., p_{j_i}, p_{j_{i+1}}\}$ have enabled writing steps that cover distinct base objects. It follows that A has an $(n-1)$-covering execution. □

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

The NSUC algorithm uses single writer multi reader (SWMR) registers and *Compare&Swap* registers (CAS). 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 a value into R. A CAS object supports the primitive $CAS(c, e, v)$ and $Read(c)$, If the value of c matches the expected value e, then $CAS(c, e, v)$ writes the new value v into c and returns *true* (the CAS succeeds). Otherwise, CAS returns $false$ and does not modify the state of the CAS (the CAS fails). $Read(c)$ returns the value in c and it does not modify its state.

In the following we describe the shared variables used by our universal construction.

- An array A of n SWMR registers. Each register contains a sequence number. 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 number that will be associated with the new state if p_i succeeds its operation. Initially, $A[j] = 0$ for $j = 1..n$.
  An array F of n SWMR registers. The process p_i writes $< sv, \sigma >$ in $F[i]$ if it has detected that it is the first process to announce its intention to define a state for the sequence number sv. σ is a pointer to the state proposed by p_i for the sequence number sv. 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 $< sv, s >$ into $OS[i]$ where s is the pointer to the new state of the shared object computed by p_i while executing its operation and sv is the associated sequence number. Initially, $OS[j] =< 0, \perp >$ for $j = 1..n$.
- A CAS register OC. It is used in case of contention to decide the new state of the object among the ones proposed by the concurrent operations. If a

process p_i detects contention, it tries to change the value of OC into a tuple $< sv, id, s >$ where id is the identifier of the process that proposes the state pointed by s and associated with the sequence number sv. id may be different than i if process p_i detects that another process p_{id} is the first one to propose a new state for sv. p_i then helps the other process to apply its operation. 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 update the CAS register OC, a process writes the sequence number stored in OC into S. Precisely, if the value of OC is $< sv, i, s >$, $S[i]$ will be set to sv. This is necessary to ensure that a process is always aware that its operation succeeded even if its operation was completed by another (helping) process. Thus, if $S[i] = sv$ process i knows that its operation which computed the state associated with sv succeeded. Initially, $S[j] = 0$ for $j = 1..n$.

At any configuration, the tuple with the highest sequence number stored either in the CAS register OC or in the array OS contains the pointer to the current state of the implemented object.

NSUC Description

In the following, unless explicitely stated, all line numbers refer to Algorithm 1.

When a process p_i wants to execute an operation *op* on an object of type T, it first gets the current state of the object and the corresponding sequence number and stores them locally in variables *state* and *seq* respectively (line 1). Then, p_i locally applies *op* to the read state (line 2). The NSUC algorithm assumes a function $APPLY_T$`(s, op, arg)` that returns the response matching the invocation of the operation *op* on a type T in a sequential execution of *op* with input *arg* applied to the state of the object pointed by s. $APPLY_T$`(s, op, arg)` also returns a pointer to the new state of the object.

If *op* is a *read-only* operation, p_i immediately returns the response (lines 3 to 5). We suppose to know a priori if an operation is *read-only*. Then, p_i checks if some other process is concurrently executing a writing operation on it. This is done by reading the other entries of the array A and looking for sequence numbers greater than or equal to $sv + 1$.

Three cases can be distinguished :

- `lines 10 to 11.` A sequence number greater than $sv + 1$ is found. This implies that some other process already decided the state for $sv + 1$, so p_i aborts.
- `lines 13 to 17.` All the sequence numbers read by p_i are smaller than $sv + 1$. Then, p_i writes its computed new state together with the associated sequence number ($< sv + 1, newState >$) into the register $F[i]$ (line 14) and checks again for concurrent operations (line 15). Consider the case where again all the sequence numbers read by p_i in the announce array A are smaller than $sv+1$ (the other case will be studied below). Any other process

competing for $sv+1$ will discover that p_i was the first process to propose a state for $sv+1$ and then it will abort its own operation, after helping p_i to complete its operation (lines 21 to 24). Finally, p_i writes its new state into the read/write register $OS[i]$ and returns the response of the operation (lines 16-17). The state of the object associated with $sv+1$ is the one proposed by p_i.

```
1  < seq, state >← STATE() ;                    //Find the object state
2  < newState, res >← APPLY_T(state, op, arg);
3  if op is read-only then
4  |  return res
5  end
6  seq ← seq + 1;
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 seq value
11 |  return ⊥
12 end
13 if seq_A < seq then                          //The process is alone
14 |  F[i] ←< seq, newState >;
15 |  if LEVEL_A(i) < seq then          //The process is still alone
16 |  |  OS[i] ←< seq, newState >;
17 |  |  return res
18 |  end
19 else                                  //There is interval contention
20 |  < id_F, newState_F >←WHOS_FIRST(seq);
21 |  if newState_F ≠ ⊥ then            //Presence of a first process
22 |  |  newState ← newState_F;
23 |  |  id_new ← id_F;
24 |  end
25 end
26 < seq_OC, id_OC, state_OC >←READ(OC);
27 while seq_OC < seq do
28 |  if (seq_OC ≠ 0) then 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 to apply operation *op* with the input *arg* on the implemented object

– `lines 20 to 35.` p_i reads $sv+1$ in one of the other entries. So, it detects that another process is concurrently trying to decide the state for this sequence number. If the detection is done on line 13, then p_i checks the presence of a process p_j competing for $sv+1$ 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 new state of the object (lines 21 to 24). In particular, since there is contention p_i will try to write state computed by p_j 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 $sv+1$. 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 15.

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

Algorithm 2: function $STATE()$

$STATE$ returns a pointer to the current state of the shared object and its sequence number.

```
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 $LEVEL_A(i)$

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

```
1 for j = 1..n do
2 |   < seq_F, σ_F > ← F[j];
3 |   if seq_F = sv then return < j, σ_F > end
4 end
5 return < 0, ⊥ >
```

Algorithm 4: function $WHOS\_FIRST(sv)$

For a given sequence number sv, $WHOS\_FIRST(sv)$ returns the tuple (j, σ) where j is the first process (if any) to propose a new state for sv and σ is a pointer to the proposed state.

```
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 $OLD\_WIN((seq_{OC}, id_{OC})$

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 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. p can recover the status of its operation checking into its location in S and then it can return the correct response.

Complexity

Let t be the worst case step complexity to perform an operation on the sequential implementation of the object (i.e. the time complexity of the function $APPLY_T$). By inspecting the pseudocode it is simple to see that the step complexity of the functions $STATE$, $LEVEL_A$ and $WHOS\_FIRST$ is in $O(n)$. Also the step complexity of the function OLD_WIN is $O(1)$. We establish that a process can repeat the loop (lines 27 to 31 of the Algorithm 1) at most n times. Thus, the step complexity of our construction is $O(n + t)$. Since the execution of every operation includes the execution of the functions $STATE$ and $APPLY_T$, the step complexity of the NSUC construction is in $\Theta(n + t)$.

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

5 Proof sketch of NSUC

In this section we sketch the ideas behind the correctness of our construction. The complete proof can be found in [4]. In the following all the line numbers refer to Algorithm 1 unless otherwise stated.

Wait-freedom. 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 number smaller than the seq value of p when executing line 27, in between the last read of OC by p and the last application of the CAS primitive to OC by p. After its CAS, q exits the loop and its operation is terminated.

The sequence numbers written in the CAS object OC and in the register $OS[i]$ $\forall i = 1..n$ are increasing. Then the next operation of q will have a sequence

number greater than or equal to seq (from line 1 and from the pseudocode of the function STATE). So, process q can prevent p to exit the loop at most once. This implies the next theorem.

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

Deterministic Abortable Object. Roughly speaking, the next theorem states that an operation that aborts does not modify the state of the object.

Theorem 3. *Let op be an operation instance executed by a process p in an execution α such that op aborts. Let sv be the sequence number computed by p at lines 1 and 6 of Algorithm 1 while executing op. The tuple with the sequence number sv and the corresponding pointer to the state computed by p will never be written either into OC or into OS.*

By inspecting the pseudocode of NSUC algorithm, a process p can abort only on line 11 or line 33. If op completes succesfully it defines the sv-th state of the object. If p aborts on line 11, it did not write into the shared register OS or any CAS. So, consider the case where process p aborts on line 33. According to line 32, one of the two following conditions has to be verified.

Either, the process p reads in OC a pointer to the state corresponding to the sequence number sv and the process identifier associated with this sequence number is different than p. This means that another operation has succeeded to define the sv-th state of the object. We complete the proof by proving that each sequence number is associated with a single state.

Or the process p reads in OC a sequence number greater than sv. We prove that if a state corresponding to a sequence number v has been written into OC So, a state $< sv, i, state >$ has been defined for the sv-th sequence number and it has been overwritten. We prove that before the overwritting of the tuple $< sv, i, state >$ in OC a process has written in $S[i]$ the sequence number sv, in order to notify process i that the state it proposed for sv has been applied. Therefore, process p aborts only if $S[p] \neq sv$. This means that the state computed by p while executing op has not been associated with sv.

Non-triviality. In its first steps, a process p executing an operation op computes sv, the sequence number associated with the state it will define if op succeeds. To compute sv, (line 1) p reads the greatest sequence number associated with a state of the object and it increments this value by one (line 6). Then, p announces its intention to define the sv-th state by writing sv in a shared register $A[p]$ (line 7). p may abort its operation op only if it detects (by reading A) an operation op' with a sequence number greater than or equal to sv. We prove that op is concurrent with op'.

Also, we prove that if op aborts, at the configuration immediately after it aborts the sv-th state has been defined. As the sequence number written in CAS OC and OS are increasing, if process p executes a new operation, this latter will be associated with a greater sequence number than sv. An operation does not

change its sequence number.Thus, for any operation op' that does not complete, process p will eventually only execute operations whose sequence number are greater than the sequence number of op'. Thus, op' cannot cause the abort of these latter operations. In particular, we prove that an operation that does not complete can cause the abort of at most two operations per process.

Theorem 4. *The universal construction NSUC is non-trivial.*

Non-trivial solo-fast. Similarly to the non-trivial property, we prove that during the execution of an operation a process applies some no-histoyless primitives only if this operation is concurrent with another one. Also, an operation op that does not complete can cause a process to apply no-histoyless primitives for only 2 consecutive operations.

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

Linearizability. For any given execution α we construct a permutation π of the high-level operations in α such that π respects the sequential specification of the object and the real-time order between operations. Since the operations that abort do not change the state of the object and return a special response *abort*, they do not impact on the response returned by the other operations and on the state computed by writing operations. Thus, in the following we discuss how to create the permutation without taking into account aborted operations; then we put aborted operations in π respecting their real-time order.

First, we order all writing operations according to the ascending order on the sequence number associated with them. Secondly, we consider each read-only operation in the order in which its reponse occurs in α. A read-only operation that returns the state of the object corresponding to the sequence number k is placed immediately before the writing operation with sequence number $k+1$ or at the end of the linearization if this latter write does not exist.

By inspecting the pseudocode it is simple to see that the total order defined by the sequence numbers respects the real-time order between writing operations. Also a read-only operation r that starts after the response of a successful writing operation w with sequence number i, will return a state of the object whose sequence number is greater than or equal to i. Thus r follows w in π. Similarly, consider two read-only operations op and op'. If op precedes op' in α, the sequence number of op' is greater than or equal to the sequence number of op, then op' is not ordered before op in π.

Theorem 6. *The universal construction 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 result 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 abort at most two operations per process. Similarly at most two operations per process use strong synchronization primitives being concurrent with a failed operation. Moreover, in case of contention our universal construction ensures that at least one writing operation succeeds to modify the object. Finally, a process that executes a read-only operation does not apply strong synchronization primitives and the operation always returns a legal response.

If t is the worst time complexity to perform an operation on the sequential object, then $\Theta(t + n)$ is the worst step complexity 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. On the other hand to prove our lower bound we consider base objects accessed via a set of historyless primitives, e.g., read/write registers. Thus, it does not imply that n CAS objects are needed to implement a non trivial solo-fast universal construction for deterministic abortable objects. The possibility to design this universal construction using $O(n)$ read/write registers but just a constant number of CAS objects is an open problem.

References

1. Afek, Y., Stupp, G., Touitou, D.: Long lived adaptive splitter and applications. Distributed Computing 15(2), 67–86 (2002)
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)
3. Attiya, H., Guerraoui, R., Hendler, D., Kuznetsov, P.: The complexity of obstruction-free implementations. J. ACM 56(4), 24:1–24:33 (2009)
4. Capdevielle, C., Johnen, C., Alessia, M.: Solo-fast universal constructions for deterministic abortable objects. Tech. Rep. 1480-14, LaBRI, Univ. de Bordeaux, France (May 2014), labri.fr/∼johnen/
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)
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)
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)
10. Herlihy, M.: Wait-free synchronization. ACM Trans. Program. Lang. Syst. 13(1), 124–149 (1991)
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)