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)

1 Introduction

In this paper we study efficient implementations for deterministic abortable objects. Proposed by Hadzilacos and Toueg [8] 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 guarantees that an aborted operation does not take effect. Operations that do not abort return a response which is legal w.r.t. the sequential specification of the object. Similarly to obstruction-free objects, the behavior of abortable objects degrades when there is contention. On the other hand, abortable objects always return the control to the caller of an operation, and when this happens the caller knows if the operation took place or not. This is an ideal behavior for shared objects [3].

Since they deal with correctness and progress separately, we expect simple and efficient algorithms to implement deterministic abortable objects. Attiya et al. proved that it is impossible to implement these 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., 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 n processes communicate through linearizable base 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 to implement deterministic abortable objects with a better space complexity than linear 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. Also, we 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].

Unfortunately, we prove that some abortable object implementation should have $\Omega(n)$ space complexity [4]. The proof of the lower bound is similar to the one presented in [3] and it is not provided in this paper for lack of space. Here we present a solo-fast universal construction for deterministic abortable objects whose space complexity is $O(n)$ if the implemented object has constant size. A *universal construction* [9] is a methodology for automatically transform any sequential object in a concurrent one.

In the following we provide the main ideas of our construction (A detailed description is in Section 2) : 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.

An implementation resulting from our universal construction is wait-free [9], linearizable [11], *non-trivial* [2], and *non-trivial solo-fast* (a process p applies some strong primitive when performing an

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

instance of operation *op*, only if *op* is concurrent with some other operation and an operation that remains incomplete does not justify the application of strong primitives by infinitely many other operations). Also it guarantees that operations that do not modify the object always return a legal response; And in case of contention, at least one writing operation succeeds to modify the object. 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 objects. Also it keeps at most $2n+1$ versions of the object. If t is the worst time complexity to perform an operation on the sequential object, then $\Theta(t+n)$ is the worst time complexity to perform an operation on the resulting 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 [3, 2, 8]. The ones proposed in [3, 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 [8]. This construction is derived from the universal construction presented in [9] and can be easily transformed into solo-fast by using the solo-fast consensus object proposed in [3]. 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 [12] 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 to make them solo-fast for abortable objects while ensuring all the good properties of our solution.

On the other hand, some of the ideas of our construction are similar to previous algorithms. The closest to our solution is the universal construction presented in [5] where operations can exit if requested by the invoking process and not already linearized. As our solution they use a sequence number to decide the order of operations. But their solution does not differentiate between read-only and writing operations. Also they use a CAS object to store the sequence number together with the process id of the corresponding operation. So the universal construction in [5] is not solo fast.

Abortable objects behave similarly to transactional memory, [10]. 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 permissiveness* and *strong progressiveness* proposed for transactional memory respectively in [13] and in [7] when conflicts are at the granularity of the entire implemented object.

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

We suppose to know if an operation *op* may change 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 input. 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.

The algorithm uses the following shared variables :

- 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 $j = 1..n$.

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). 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). 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.

```
   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 return res end
 4 seq ← seq + 1 ;                                                          //New sequence
 5 A[i] ← seq ;                                          //The process announces its intention
 6 id_new ← i;
 7 seq_A ← LEVEL_A(i);
 8 if seq_A > seq then return ⊥ end                  //A state is already decided for this sequence
 9 if seq_A = seq then                                          //There is an interval contention
10 |  < id_F, newState_F >←WHOS_FIRST(seq);
11 |  if newState_F ≠ ⊥ then newState ← newState_F; id_new ← id_F; end        //Presence of a first
   |  process
12 else
13 |  F[i] ←< seq, newState >;
14 |  if LEVEL_A(i) < seq then OS[i] ←< seq, newState >; return res end    //The process is alone
15 end
16 < seq_OC, id_OC, state_OC >←READ(OC);
17 while seq_OC < seq do
18 |  OLD_WIN(seq_OC, id_OC);
19 |  CAS(OC, < seq_OC, id_OC, state_OC >, < seq, id_new, newState >);
20 |  < seq_OC, id_OC, state_OC >←READ(OC);
21 end
22 if (seq_OC = seq ∧ id_OC ≠ i) ∨ (seq_OC > seq ∧ READ(S[ij]) ≠ seq) then res ← ⊥; end
23 return res

   Code for the function STATE(), which returns the current state of the shared object and
   its sequence :
24 seq_max ← 0; σ_max ← ⊥ ;
25 for j = 1..n do
26 |  < seq_OS, σ >← OS[j];
27 |  if seq_OS > seq_max then seq_max ← seq_OS; σ_max ← σ; end
28 end
29 < seq_OC, id_OC, σ_OC >←READ(OC);
30 if seq_OC < seq_max then return < seq_max, σ_max > end
31 return < seq_OC, σ_OC >

   Code for the function LEVEL_A(i), which returns the highest sequence written into A by a
   process other than p_i :
32 seq_max ← 0;
33 for j = 1..n | j ≠ i do
34 |  seq_A ← A[j];
35 |  if seq_max < seq_A then seq_max ← seq_A; end
36 end
37 return seq_max

   Code for the function WHOS_FIRST(seq), which 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 :
38 for j = 1..n do
39 |  < seq_F, σ_F >← F[j];
40 |  if seq = seq_F then return < j, σ_F > end
41 end
42 return < 0, ⊥ >

   Code for the function OLD_WIN(seq_OC, id_OC), which ensures that a slow process p is aware
   if 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.
43 seq_S ← READ(S[id_OC]);
44 if seq_OC > seq_S then CAS(S[id_OC], seq_S, seq_OC); end
```

Algorithm 1: NSUC - Code for process p_i

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., 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)
4. Capdevielle, C., Johnen, C., Milani, A.: Solo-fast universal constructions for deterministic abortable objects. Tech. Rep. 1480-14, LaBRI (may 2014)
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. 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)
8. 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
9. Herlihy, M.: Wait-free synchronization. ACM Trans. Program. Lang. Syst. 13(1), 124–149 (1991), http://doi.acm.org/10.1145/114005.102808
10. 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)
11. Herlihy, M., Wing, J.M.: Linearizability: A correctness condition for concurrent objects. ACM Transactions on Programming Languages and Systems 12(3), 463–492 (1990)
12. 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)
13. 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)