Concurrent Systems: Hybrid Object Implementations and Abortable Objects

Michel Raynal[1,2,3(✉)]

[1] Institut Universitaire de France, Paris, France
[2] IRISA, Université de Rennes, Rennes, France
[3] Department of Computing, Polytechnic University, Hong Kong, China
raynal@irisa.fr

Abstract. As they allow processes to communicate and synchronize, concurrent objects are, de facto, the most important objects of concurrent programming. This paper presents and illustrates two important notions associated with concurrent objects. The first one, which is related to their implementation, is the notion of a hybrid implementation. The second one, which is related to their definition, is the notion of an abortable object.

1 Introduction

Concurrent Objects: On the Classical Side. An *object* is a (passive) computing entity providing processes with operations. Only these operations are visible from outside the object. Said differently, the internal representation, of an object remains invisible to the processes. Hence, an object is an abstraction. An object is defined by a specification, which states the properties defining all its correct behaviors. Those are usually captured by the set of the allowed sequences on operation invocations. It appears that the object notion was introduced a long time ago (it seems that its very first appearance dates back in 1967, in the language SIMULA 67 [21]).

The first object, specific to the domain of concurrent programming, seems to be the *semaphore* [3,9]. It is a counting object whose value has to never become negative. Hence, processes can decrease and increase it as long they maintain invariant the fact that it remains non-negative. Then, more sophisticated object constructs have been introduced to cope with concurrent objects, such the concept of a *monitor* [4,20]. In nearly all cases, these language constructs reduce concurrency to sequential computing (they basically ensure that the object operations are executed in mutual exclusion).

Concurrent Objects: The World is Changing. Concurrency in multiprocessors (e.g., multicore) is *real* concurrency. It follows that the concurrency concepts and techniques used to cope with multiplexing or interrupt handling are no longer appropriate, and must be revisited to address the new computing world. As expressed in [15]: "Changes in technology can have far-reaching effects on theory. [...] After decades of being respected but not taken seriously, research

J.L. Träff et al. (Eds.): Euro-Par 2015, LNCS 9233, pp. 3–15, 2015.
DOI: 10.1007/978-3-662-48096-0_1

on multiprocessor algorithms and data structures is going mainstream" (see also [18,29,34]).

Among the most important concepts that have been introduced there is the notion of progress conditions suited to systems where processes are asynchronous and can fail by crashing. These notions are obstruction-freedom [16], non-blocking [19], and wait-freedom [14]. This has motivated researchers to re-think the implementation of concurrent data structures to exploit the benefit of new architectures (e.g. [2,7,8,26,27,35,36] to cite a few).

Concurrent Objects: The Topics Addressed in the Paper. This paper first defines (Sect. 2) basic computing models, which allow us to reason on concurrent objects. Their main characteristics lies in the hardware operations the processes can use, the asynchrony of the processes, and the fact that failures can occur or not.

Then the paper presents the notion of a hybrid implementation of a concurrent object (Sect. 3). Roughly speaking, a hybrid implementation is such that the algorithms implementing the object operations do not use locks in "good circumstances", those being defined statically or dynamically. It follows that in concurrency-free patterns, locks are not used.

Finally (Sect. 4), the paper considers the notion of an abortable object, and illustrates it with a non-blocking abortable stack. An abortable object allows operations to return a default value $\perp$ when operation invocations are concurrent.

The paper is an introductory paper to concurrent objects, addressing hybrid implementation and abortable objects, which are only two facets of concurrent objects. The reader will find more developments on concurrent programming objects in [18,29,33].

2 Computing Models, Objects, and Progress Conditions

2.1 Basic Read/Write Model and Enriched Models

Basic Read/Write Model. The basic read/write model consists of n sequential asynchronous deterministic processes, denoted p_1, ..., p_n, which communicate by reading and writing atomic registers only.

Asynchronous means each process proceeds to its own speed, which is not known by other processes, and can be arbitrary and vary with time. *Deterministic* means that the behavior of a process is entirely determined from its initial state, the algorithm it executes, and the sequence of values read from atomic registers. *Atomic* means that, for each register, the read and write operations appear as if they had been invoked sequentially, each abstracted as a point of the time line occurring between its start and its end [19,23].

This computation model, where there are no failures, is denoted $\mathcal{ARW}_n$.

Crash Failures. The most common failures studied in multicore distributed computing are process crash failures and Byzantine failures. Here we consider

only process crash failures. Such a failure occurs when a process halts unexpectedly. Before crashing it executes correctly its algorithm, and after it crashed, a process remains crashed forever.

Let t be the maximal number of processes that may crash; t is a model parameter and the corresponding model is called *t-resilient* model. The asynchronous read/write model in which all processes, except one, may crash is called *wait-free* model. Hence, "wait-free model" is synonym of "$(n-1)$-resilient model".

This crash-prone computation model is denoted $\mathcal{ARW}_n[\emptyset]$. When enriched with hardware-provided objects of some type $TYPE$ (whose aim is to allow processes to communicate), the corresponding system model is denoted $\mathcal{ARW}_n[TYPE]$.

Enriched Model. While, from a computability point of view, $\mathcal{ARW}_n[\emptyset]$ has the same power as a Turing machine, this is no longer the case for $\mathcal{ARW}_{n,t}[\emptyset]$ which is strictly weaker than $\mathcal{ARW}_n[\emptyset]$ as soon as only even one process may crash (i.e., for any $t > 0$) [10,14,17,25,30,33].

The situation is different as soon as processes can use hardware-provided synchronization objects stronger than atomic read/write registers, such as test&set objects, compare&swap objects, or LL/SC objects, to cite a few. The crash-prone computation model, enriched with objects of type $TYPE$, is denoted $\mathcal{ARW}_{n,t}[TYPE]$.

It was shown by Herlihy [14] that the computability power of such objects can be measured with the notion of a *consensus number*. The greater this number, the stronger the object. An infinite hierarchy of objects suited to the wait-free model has been exhibited, where it is shown that the consensus number of registers is 1, the one of test&set objects is 2, while the one of compare&swap or LL/SC objects is infinite. Hence, the model $\mathcal{ARW}_{n,n-1}$[Compare&swap] is computationally stronger than $\mathcal{ARW}_{n,n-1}$[Test&set], which itself is stronger than $\mathcal{ARW}_{n,n-1}[\emptyset]$ (see, e.g., [14,29]).

2.2 Concurrent Objects

Definition. A *concurrent* object (sometimes also called *shared* object) is an object that can be accessed by several processes, simultaneously or not.

We consider here the subset of concurrent objects defined by a sequential specification on total operations. An operation is *total* if it always returns a result, whatever the state of the object (e.g., the operation remove() applied to an empty queue is not allowed to wait until an element is added to the queue; it must always terminate, for example returning the control value *empty*). *Sequential specification* means that the correct behaviors of an object can be expressed by traces on its operation invocations.

One-Shot vs Multi-shot. An object is *one-shot* if it has only one operation and each process is allowed to invoke this operation at most once. Otherwise, the object is *multi-shot*. As an example, a consensus object is one-shot, while a set object or a stack are multi-shot objects.

Consistency Condition. The most familiar consistency condition considered for concurrent objects is *atomicity* [23], also called *linearizability* [19]. It states that it must be possible to totally order the operations issued on each object in such a way that (a) this total order respects the occurrence order of non-concurrent operations, and (b) the resulting sequence of operations belongs to the specification of the object.

An important property of linearizability, which motivates its practical consideration, lies in its *composability* dimension (also called *locality*) [19], namely, linearizable objects compose for free. This means that if we have two linearizable objects $O1$ and $O2$ (whose implementations are independent) then the composed object $\langle O1, O2\rangle$ is also linearizable, and this is obtained for free, i.e., without additional implementation cost. (Intuitively, this comes from the fact that linearizability respects the occurrence order of non-concurrent operations).

It is important to notice that other consistency conditions such as *sequential consistency* [22] (or non-strict serializability encountered in databases [28]) are not composable. This means that, to obtain a sequentially consistent composed object $\langle O1, O2\rangle$ from two sequentially consistent objects $O1$ and $O2$, the implementation of both $O1$ and $O2$ has to be modified, each one must cooperate with the other one to ensure that the composite object is sequentially consistent [29].

2.3 Progress Conditions for Object Operations

Classical Progress Conditions. The classical progress conditions encountered in the implementation of concurrent objects are *deadlock-freedom* and *starvation-freedom*. The first one captures the point of view of the object (service), namely, if processes concurrently invoke operations, at least one process succeeds. The second one captures the point of view of the processes (clients), namely, if any process invokes an operation, it eventually executes it. These progress conditions are usually implemented with locking mechanisms. Trivially, starvation-freedom $\Rightarrow$ deadlock-freedom.

Locks in the Presence of Failures. It is important to notice that locks cannot be used in the system model $\mathcal{ARW}_{n,t}[\emptyset]$. This is due to the following reason. If a process p obtains a lock on an object and crashes before unlocking it, due to asynchrony, no other process can distinguish the case where p crashes and the case where p is slow. Hence, in an asynchronous crash-prone system, locks may prevent processes from progressing.

Progress Conditions Suited to Net Effect of Asynchrony and Process Crashes. Three progress conditions have been proposed to cope with the net effect of asynchrony and process crashes. Actually, a process crash can be seen as if the corresponding process was pausing during a "very" long period of time, during which the non-faulty processes must progress despite its absence of progress.

Obstruction-Freedom [16] is the weakest progress conditions (from a progress point of view). It states that a non-faulty process, that invokes an object operation, is required to terminate it, if it executes alone during a "long enough"

period. ("Long enough" means it has enough time to terminate its operation, without being bothered by other processes). Hence, obstruction-freedom allows concurrent operations to never terminate.

Non-blocking is a stronger progress condition [19]. It states that, whatever the concurrency among operation invocations, at least one of the concurrent invocations terminate. As one can see, this is nothing else than deadlock-freedom in a context where locks are forbidden (also called mutex-free context).

Finally, *wait-freedom* is the strongest progress condition [14]. It requires that, until it possibly crashes, and whatever the behavior of the other processes, all the operations issued by a process terminate.

Trivially, wait-freedom $\Rightarrow$ non-blocking $\Rightarrow$ obstruction-freedom. Of course, the previous three mutex-free progress conditions remain meaningful in the classical failure system model $\mathcal{ARW}_n[\emptyset]$.

Where is the Difficulty. As previously indicated, locks cannot be used when one has to cope with asynchrony and failures. Hence, *mutex-free* solutions have to be found [29].

The main difficulty when one wants to implement a concurrent object whose operations have to satisfy one of obstruction-freedom, non-blocking, or wait-freedom, comes from the fact that there is no way to prevent several processes to simultaneously access the internal representation of the object. This is true even for the weak obstruction-freedom progress condition. As a simple example, let us consider two processes that, invoking the operation $S.\mathsf{pop}()$ on a stack S, access simultaneously its internal representation, and then one of them pauses during a long enough period that allows the other process to terminate. Both the returned values must be correct.

According to the high level object that has to be built, solving this issue may require basic objects whose computational power is stronger than read/write registers. In some cases, the power required is the one provided by the most powerful objects (such as compare&swap) when considering the consensus number hierarchy.

3 Hybrid Implementation of a Concurrent Object

3.1 The Notion of a Hybrid Implementation of a Concurrent Object

Definition. The idea that underlies the notion of a hybrid implementation is that locks are expensive, and consequently their use must be prevented in some circumstances.

Given a concurrent object O, an *hybrid implementation* of O is an implementation that merges lock-based code and mutex-free code in the implementation of the operations of O. This notion has been introduced in an explicit way in [29]. As locks can be used, these implementations are for failure-free systems. (Said differently, this means that, in the presence of failures, the system may stop progressing in configurations where a process crashes while holding a lock.)

Static Hybrid vs Dynamic Hybrid Implementation. Two types of hybrid implementations can be distinguished.

- *Static* hybrid implementation. In this case, the operations on the object are statically divided into subsets: the ones whose implementation can use locks, and the others whose implementation cannot use locks.
- *Dynamic* hybrid implementation. In this case, whatever the operation, its implementation cannot use locks in "favorable circumstances". Those are defined according to the object, the context in which it is used, the aplication features, etc.

3.2 Example 1: Static Hybrid Implementation of a Set Object

Set Object A concurrent set object S is defined by three operations:

- S.add(v) adds v to the set S and returns `true` if v was not in the set. Otherwise it returns `false`.
- S.remove(v) suppresses v from the set S and returns `true` if v was in the set. Otherwise it returns `false`.
- S.contain(v) returns `true` if v belongs to the set. Otherwise it returns `false`.

In a lot of applications using a set object, the number of invocations of S.contain() outperforms the number of invocations of S.add() and S.remove(). This is, for example, the case of dictionary-like objects. In such a context, for efficiency reasons, we want to have an implementation of S.contain() that (a) is mutex-free (it does not use locks), and (b) always terminates. Said more compactly, the algorithm implementing S.contain() has to be wait-free. Differently, the algorithms implementing the operations S.add() and S.remove() may use locks. These operations are required to be only deadlock-free. Moreover, to allow them to be as concurrent as possible, a lock is associated with each element of the set, and a process can simultaneously hold locks on at most two elements.

An hybrid implementation of such a concurrent set has been proposed in [13] and proved correct in [6] (see also [29] for a pedagogical presentation). This implementation is list-based. It assumes that the elements of the set are totally ordered, they have a smallest element and a greatest element, and there is finite number of elements between any two elements.

3.3 Example 2: Dynamic Hybrid Implementation of a Double-Ended Queue

A dynamic hybrid implementation of a double-ended queue (in short, dequeue) is presented in [16]. The "favorable circumstances" are when there is no concurrency. The main difficulty that this implementation has to solve occurs when a process started executing an operation in a no-concurrency context (hence it uses no lock) and, while it is executing its operation, another process issue a conflicting operation. This implementation considers the enriched system model $\mathcal{ARW}_n$[Compare&swap]. A version of it, suited to the system model $\mathcal{ARW}_n$[LL/SC] is described in [29,34].

Due to page limitation, the reader will consult [16,29] for a full presentation of these implementations.

3.4 Example 3: Dynamic Hybrid Implementation of a Consensus Object

Binary Consensus Object. Such an object C is a one-shot object that provides the processes with a single operation denoted C.propose() ans returns a value, called "decided value". Only the values 0 and 1 can be proposed. The object is defined by the following properties.

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

Favorable Circumstances. Here "favorable circumstances" concern two different cases. The first is when all the processes that invoke C.propose(), propose the same value. The second is when an invocation of propose() executes in a concurrency-free context.

When a favorable circumstance occurs, no lock has to be used. This means that an invocation of propose(v) is allowed to use an underlying lock only if (a) the other value $(1-v)$ was previously or is currently proposed, and (b) there are concurrent invocations. Hence, from a lock point of view, the notion of conflict is related to both concurrency and proposed values.

Dynamic Hybrid Implementation: Internal Representation of the Object. The implementation that follows is from [34]. The internal representation of the consensus object is made up of the following atomic read/write registers, plus a lock:

- $PROPOSED[0..1]$, which is an array of two Boolean registers, both initialized to *false*. The atomic register $PROPOSED[v]$ is set to *true* to indicate that a process has proposed value v.
- $DECIDED$, which is an atomic register whose domain is $\{\bot, 0, 1\}$. Initialized to $\bot$, it is eventually set to the value that is decided and never the value which is not decided.
- AUX, which is an atomic register whose domain and initial value are the same as for $DECIDED$.
- $LOCK$, which is a starvation-free lock used to solve conflicts (if any).

Dynamic Hybrid Implementation: Algorithm. This algorithm is described in Fig. 1. A process decides when it executes the statement return(val), where val is the value it decides.

When a process p invokes propose(v), it first indicates that v was proposed, and writes it into AUX if this register is still equal to $\bot$ (line 01). Let us notice that, if several processes proposing different values concurrently read $\bot$ from AUX, each writes its proposed value in AUX.

Then, process p checks if the other binary value $(1 - v)$ was proposed by another process (line 02). If it is not the case, p writes v into $DECIDED$ (line 03), and assuming that no other process has written a different value into $DECIDED$

```
operation C.propose(v) is
(01)  PROPOSED[v] ← true; if (AUX = ⊥) then AUX ← v end if;
(02)  if (¬PROPOSED[1 − v])
(03)     then DECIDED ← v
(04)     else if (DECIDED = ⊥)
(05)             then LOCK.acquire_lock();
(06)                  if (DECIDED = ⊥) then DECIDED ← AUX end if;
(07)                  LOCK.release_lock()
(08)          end if;
(09)  end if;
(10)  return(DECIDED)
end operation.
```

Fig. 1. A dynamic hybrid implementation of a binary consensus object in $\mathcal{ARW}_n$ [LOCK] [34]

in the meantime, it decides the value stored in *DECIDED* (line 10). If the other value was proposed there is a conflict. Process p then decides the value kept in *DECIDED* if there is one (lines 04 and 10). If there is no decided value, the conflict is solved with the help of the lock (lines 05–07). Process p assigns the current value of AUX to *DECIDED* if that register was still equal to $\perp$ when it read it (lines 06) and p finally decides the value kept in *DECIDED*. Proofs can be found in [29,34].

4 Abortable Concurrent Objects

4.1 The Notion of a Concurrent Abortable Object

In practice, conflicts are rare in a lot of applications. So the idea is here, not only to forbid locks at the implementation level, but, at the semantics/interface level, allow a process, that invokes an object operation, to return a predefined default value $\perp$ (abort) in specific circumstances, namely in the presence of concurrency[1].

Hence, the meaning of $\perp$ is "the operation has not been executed because the invocation occurred in a concurrency context". Moreover, if we do not consider the operation invocations that return $\perp$, an abortable object behaves as described by its sequential specification.

[1] In some sense, the origin of abortable objects can be found in Lamport's fast mutex algorithm [24]. This paper presents a mutual exclusion algorithm which allows a process to take a *fast path* to access the critical section when there is no concurrency. This fast path requires only five accesses to atomic read/write registers, and is consequently independent of the total number of processes. When there is concurrency, the number of accesses to atomic read/write registers is $O(n)$. This algorithm was the starting point of the design of *time-adaptive* algorithms. The time complexity of such an algorithm A is $O(f(d)) \leq O(f(n))$, where $d \in [1..n]$ is the concurrency degree at the time where the object operation implemented by A is executed.

This notion of an abortable object, introduced in [12,29], has not to be confused with a close (but different) notion introduced in [1]. In this paper, when an operation returns $\perp$, the invoking process learns that its call occurred in a concurrency context, but it does know if the operation was executed or not.

4.2 Example: A Non-blocking Abortable Stack in $\mathcal{ARW}_{n,n-1}$[Compare&swap]

The implementation of a non-blocking abortable stack presented below is from [32]. It is based on compare&swap objects.

Compare&swap Object and the ABA Problem. A compare&swap object X is an atomic register that can be read, and can be written by a hardware-provided operation called compare&swap(). This operation is a conditional write, which has two input parameters (denoted *old* and *new*), and returns a Boolean value. Its effect can be described as follows:

X.compare&swap(old, new) **is**
if $(X = old)$ **then** $X \leftarrow new$; return($true$) **else** return($false$) **end if**.

When using compare&swap(), a process p_i usually does the following. It first reads the atomic register X (obtaining its current value a), then executes statements (possibly involving accesses to the shared memory) and finally updates X to a new value c only if X has not been modified by another process since it was read by p_i. To that end, p_i invokes X.compare&swap(a, c).

Unfortunately, the fact that this invocation returns *true* to p_i does not allow p_i to conclude that X has not been modified since the last time it read it. This is because, between the read of X and the invocation X.compare&swap(a, c) both issued by p_i, X could have been updated twice, first by a process p_j that successfully invoked X.compare&swap(a, b), and then by another process p_k that successfully invoked X.compare&swap(b, a), thereby restoring the value a to X. This is called the ABA problem.

This problem can be solved by associating sequence numbers with each value that is written (see [29]). Hence, in the previous scenario, the read of X by p_i would have returned a pair $\langle a, sn\rangle$. Then, $X = \langle a, sn+2\rangle$ after the the successful invocations issued by p_j and p_k. Hence, the X.compare&swap($\langle a, sn\rangle, c$) cannot be successful.

Abortable Stack: Operations. The stack operations are denoted push(v) (where v is the value to be added at the top of the stack) and pop(). The stack is a bounded stack: it can contain at most k values. If the stack is full, push(v) returns the control value *full*, otherwise v is added at the top of the stack and the control value *done* is returned. The operation pop() returns the value that is at the top of the stack (and suppresses it from the stack), or the control value *empty* if the stack is empty. Both operations may return $\perp$ in the presence of concurrency.

Non-blocking Abortable Stack. As the stack must be non-blocking, even in the presence of concurrency, at least one operation does not return $\perp$.

Non-blocking Abortable Stack: Internal Representation. The stack is implemented with an atomic register denoted TOP and an array of $k+1$ atomic registers denoted $STACK[0..k]$. These registers can be read and can be modified only by using the compare&swap() primitive.

- TOP has three fields that contain an index (to address an entry of $STACK$), a value, and a counter. It is initialized to $\langle 0, \perp, 0\rangle$.
- Each atomic register $STACK[x]$ has two fields: the field $STACK[x].val$, which contains a value, and the field $STACK[x].sn$, which contains a sequence number (used to prevent the ABA problem as far as $STACK[x]$ is concerned). $STACK[0]$ is a dummy entry initialized to $\langle \perp, -1\rangle$. Its first field always contains the default value $\perp$. As far as the other entries are concerned, $STACK[x]$ $(1 \leq x \leq k)$ is initialized to $\langle \perp, 0\rangle$.

The array $STACK$ is used to store the contents of the stack, and the register TOP is used to store the index and the value of the element at the top of the stack. The contents of TOP and $STACK[x]$ are modified with the help of the conditional write operation compare&swap() (which, with the help of sequence numbers, is used to prevent erroneous modifications of the stack internal presentation).

A Non-blocking Abortable Stack: The Algorithm. The implementation is *lazy* in the sense that a stack operation assigns its new value to TOP and leaves the corresponding effective modification of $STACK$ to the next stack operation. Hence, while on the one hand a stack operation is lazy, on the other hand it has to help terminate the previous stack operation (as far as the internal representation of the stack is concerned) (Fig. 2).

When a process p_i invokes push(v), it first reads the content of TOP (which contains the last operation on the stack) and stores its three fields in its local variables $index$, $value$, and $seqnb$ (line 01).

Then, p_i calls the internal procedure help$(index, value, seqnb)$ to help terminate the previous stack operation (line 02). That stack operation (be it a push() or a pop()) is required to write the pair $\langle value, seqnb\rangle$ into $STACK[index]$. To that end, p_i invokes $STACK[index]$.compare&swap.(old, new) with the appropriate values old and new so that the write is executed only if not yet done (lines 15–16).

After its help (which was successful if not yet done by another stack operation) to move the content of TOP into $STACK[index]$, p_i returns *full* if the stack is full (line 03). If the stack is not full, it tries to modify TOP so that it registers its push operation. This invocation of TOP.compare&swap() (line 06) succeeds if no other process modified TOP since it was read by p_i at line 01. If it succeeds, TOP takes its new value and push(v) returns the control value *done* (line 06). Otherwise, p_i returns $\perp$ (line 07).

The triple of values to be written in TOP at line 06 is computed at lines 04–05. Process p_i first computes the last sequence number sn_of_next used in $STACK[index+1]$ and then defines the new triple, namely $newtop = \langle index + 1, v, sn\_of\_next+1\rangle$, to be written first in TOP and, later, in $STACK[index+1]$

```
operation push(v) is
(01)  (index, value, seqnb) ← TOP;
(02)  help(index, value, seqnb);
(03)  if (index = k) then return(full) end if;
(04)  sn_of_next ← STACK[index + 1].sn;
(05)  newtop ← ⟨index + 1, v, sn_of_next + 1⟩;
(06)  if TOP.compare&swap(⟨index, value, seqnb⟩, newtop)
(07)        then return(done) else return(⊥) end if
end operation.

operation pop() is
(08)  (index, value, seqnb) ← TOP;
(09)  help(index, value, seqnb);
(10)  if (index = 0) then return(empty) end if;
(11)  belowtop ← STACK[index − 1];
(12)  newtop ← ⟨index − 1, belowtop.val, belowtop.sn + 1⟩;
(13)  if TOP.compare&swap(⟨index, value, seqnb⟩, newtop)
(14)        then return(value) else return(⊥) end if
end operation.

internal procedure help(index, value, seqnb):
(15)  stacktop ← STACK[index].val;
(16)  STACK[index].compare&swap(⟨stacktop, seqnb − 1⟩, ⟨value, seqnb⟩)
end procedure.
```

Fig. 2. A non-blocking abortable stack in $\mathcal{ARW}_{n,n-1}$[Compare&swap] [32]

thanks to the help provided by the next stack operation ($sn\_of\_next + 1$ is used to prevent the ABA problem).

5 Conclusion

Considering concurrent objects, the aim of this paper was to present the notion of an hybrid implementation of such objects, and the notion of an abortable object. To this end, it first introduced fundamental notions associated with concurrent objects (namely, consistency conditions and progress conditions). Then, after having defined the notions of an hybrid implementation and an abortable object, it illustrated them with appropriate examples.

The reader interested in concurrent programming can consult the following textbooks devoted to concurrent objects, where are presented concurrency-related concepts, algorithms, techniques, and numerous object implementations [18,29,33] (parts of this paper are from [29]). A more general and sophisticated notion, related to concurrent objects, is the one of a *universal construction* for concurrent objects. This important topic, focusing on *universality in the presence of concurrency, asynchrony and process crash failures*, introduced in [14], is addressed in the previous textbooks, and in [5,11,31].

References

1. Aguilera, M.K., Frolund, S., Hadzilacos, V., Horn, S.L., Toueg, S.: Abortable and query-abortable objects and their efficient implementation. In: Proceedings of 26th ACM Symposium on Principles of Distributed Computing (PODC 2007), pp. 23–32. ACM Press (2007)
2. Afek, Y., Attiya, H., Dolev, D., Gafni, E., Merritt, M., Shavit, N.: Atomic snapshots of shared memory. J. ACM **40**(4), 873–890 (1993)
3. Hansen, P.B.: The Architecture of Concurrent Programs. Prentice Hall, Upper Saddle River (1977)
4. Hansen, P.B. (ed.): The Origin of Concurrent Programming, p. 534. Springer, New York (2002)
5. Capdevielle, C., Johnen, C., Milani, A.: Solo-fast universal constructions for deterministic abortable objects. In: Kuhn, F. (ed.) DISC 2014. LNCS, vol. 8784, pp. 288–302. Springer, Heidelberg (2014)
6. Colvin, R., Groves, L., Luchangco, V., Moir, M.: Formal verification of a lazy concurrent list-based set algorithm. In: Ball, T., Jones, R.B. (eds.) CAV 2006. LNCS, vol. 4144, pp. 475–488. Springer, Heidelberg (2006)
7. Crain, T., Gramoli, V., Raynal, M.: A contention-friendly binary search tree. In: Wolf, F., Mohr, B., an Mey, D. (eds.) Euro-Par 2013. LNCS, vol. 8097, pp. 229–240. Springer, Heidelberg (2013)
8. Crain, T., Gramoli, V., Raynal, M.: No hot spot non-blocking skip list. In: Proceedings of 33rd International Conference on Distributed Computing Systems (ICDCS 2013), pp. 196–205. IEEE Press (2013)
9. Dijkstra, E.W.D.: Cooperating sequential processes. In: Genuys, F. (ed.) Programming Languages, pp. 43–112. Academic Press, New York (1968)
10. Fischer, M.J., Lynch, N.A., Paterson, M.S.: Impossibility of distributed consensus with one faulty process. J. ACM **32**(2), 374–382 (1985)
11. Gafni, E., Guerraoui, R.: Generalized universality. In: Katoen, J.-P., König, B. (eds.) CONCUR 2011. LNCS, vol. 6901, pp. 17–27. Springer, Heidelberg (2011)
12. Hadzilacos, V., Toueg, S.: On deterministic abortable objects. In: Proceedings of 32nd ACM Symposium on Principles of Distributed Computing (PODC 2013), pp. 4–12. ACM Press (2013)
13. Heller, S., Herlihy, M.P., Luchangco, V., Moir, M., Scherer, W.I.I.I., Shavit, N.: A lazy concurrent list-based algorithm. Parallel Process. Lett. **17**(4), 411–424 (2007)
14. Herlihy, M.P.: Wait-free synchronization. ACM Trans. Program. Lang. Syst. **13**(1), 124–149 (1991)
15. Herlihy, M.P., Luchangco, V.: Distributed computing and the multicore revolution. ACM SIGACT News **39**(1), 62–72 (2008)
16. Herlihy, M.P., Luchangco, V., Moir, M.: Obstruction-free synchronization: double-ended queues as an example. In: Proceedings of 23th International IEEE Conference on Distributed Computing Systems (ICDCS 2003), pp. 522–529. IEEE Press (2003)
17. Herlihy, M.P., Rajsbaum, S., Raynal, M.: Power and limits of distributed computing shared memory models. Theor. Comput. Sci. **509**, 3–24 (2013)
18. Herlihy, M., Shavit, N.: The Art of Multiprocessor Programming, p. 508. Morgan Kaufmann, Burlington (2008). ISBN 978-0-12-370591-4
19. Herlihy, M.P., Wing, J.M.: Linearizability: a correctness condition for concurrent objects. ACM Trans. Program. Lang. Syst. **12**(3), 463–492 (1990)

20. Hoare, C.A.R.: Monitors: an operating system structuring concept. Commun. ACM **17**(10), 549–557 (1974)
21. https://en.wikipedia.org/wiki/Simula
22. Lamport, L.: How to make a multiprocessor computer that correctly executes multiprocess programs. IEEE Trans. Comput. C **28**(9), 690–691 (1979)
23. Lamport, L.: On interprocess communication. Part I: basic formalism. Distrib. Comput. **1**(2), 77–85 (1986)
24. Lamport, L.: Fast mutual exclusion. ACM Trans. Comput. Syst. **5**(1), 1–11 (1987)
25. Loui, M., Abu-Amara, H.: Memory Requirements for Agreement among Unreliable Asynchronous Processes. Advances in Computing Research, pp. 163–183. JAI Press, Greenwich (1987)
26. Michael, M.M., Scott, M.L.: Simple, fast and practical blocking and non-blocking concurrent queue algorithms. In: Proceedings of 15th International ACM Symposium on Principles of Distributed Computing (PODC 1996), pp. 267–275. ACM Press (1996)
27. Moir, M., Nussbaum, D., Shalev, O., Shavit, N.: Using elimination to implement scalable and lock-free FIFO queues. In: Proceedings of 17th ACM Symposium on Parallelism in Algorithms and Architectures (SPAA 2005), pp. 253–262. ACM Press (2005)
28. Papadimitriou, C.H.: The serializability of concurrent database updates. J. ACM **26**(4), 631–653 (1979)
29. Raynal, M.: Concurrent Programming: Algorithms, Principles, and Foundations, p. 530. Springer, Heidelberg (2013). ISBN 978-3-642-32026-2
30. Raynal, M.: What can be computed in a distributed system? In: Bensalem, S., Lakhneck, Y., Legay, A. (eds.) From Programs to Systems. LNCS, vol. 8415, pp. 209–224. Springer, Heidelberg (2014)
31. Raynal, M., Stainer, J., Taubenfeld, G.: Distributed universality. In: Aguilera, M.K., Querzoni, L., Shapiro, M. (eds.) OPODIS 2014. LNCS, vol. 8878, pp. 469–484. Springer, Heidelberg (2014)
32. Shafiei, N.: Non-blocking array-based algorithms for stacks and queues. In: Garg, V., Wattenhofer, R., Kothapalli, K. (eds.) ICDCN 2009. LNCS, vol. 5408, pp. 55–66. Springer, Heidelberg (2008)
33. Taubenfeld, G.: SynchroNization Algorithms and Concurrent Programming, p. 423. Pearson Education/Prentice Hall, Upper Saddle Rive (2006). ISBN 0-131-97259-6
34. Taubenfeld, G.: Contention-sensitive data structures and algorithms. In: Keidar, I. (ed.) DISC 2009. LNCS, vol. 5805, pp. 157–171. Springer, Heidelberg (2009)
35. Tsigas, Ph., Zhang, Y.: A simple, fast and scalable non-blocking concurrent FIFO queue for shared memory multiprocessor systems. In: Proceedings of 13th ACM Symposium on Parallelism in Algorithms and Architectures (SPAA 2001), pp. 134–143. ACM Press (2001)
36. Valois, J.D.: Implementing lock-free queues. In: Proceedings of 7th International Conference on Parallel and Distributed Computing Systems (PDCS 1994), pp. 64–69. IEEE Press (1994)