Chapter 2

Wait-freedom: A Progress Property for Shared Object Implementations

October 1, 2007

1 Introduction

The previous chapter focused on the *atomicity* property of shared objects. Such a property stipulates that certain behaviors of the implemented object should be precluded, namely those that do not hide concurrent operation executions on the same object. Atomicity is a *safety* property. It states what should *not* happen in an execution involving processes and shared objects (namely, an execution that is not linearizable must never happen). Clearly, this could be achieved by a trivial algorithm that would block every operation of the object to be implemented, i.e., one that would never return any result.

Intuitively, one would also require that the implemented shared object indeed performs its operations when it is asked to do so by a process making use of that object. In other words, one would also require that the algorithm implementing the object satisfies some *progress* property (also called *liveness* property). Clearly, this also depends on the process invoking the operation. For instance, if a process invokes an operation and immediately crashes, then it makes little sense to require that the process obtains indeed a reply matching its invocation. In fact, one might require that the shared object satisfies some termination property, provided that the process invoking an operation on the shared object (a) indeed executes the algorithm implementing that operation and (b) the operating system indeed schedules *enough steps* of that process. For instance, one might require that if a process invokes an operation and keeps correctly executing steps of the algorithm implementing the operation, then the operation eventually terminates and the process obtains a response. Hence, studying progress requires to zoom into the way processes execute the algorithms implementing the shared object operations.

As discussed in Chapter 1, when implementing atomic objects, it is enough to consider each object separately from the other objects. This is because the atomicity consistency criterion is a local property. In this chapter, we will focus on a single object and discuss how to implement it from underlying base objects while ensuring certain progress properties. We will go over several such properties, the strongest of these being wait-freedom.

2 Implementation

2.1 high level object and low level object

To distinguish the shared object to be implemented from the underlying objects used in the implementation, we typically talk about a *high level* object and underlying *low level* objects. (The latter are sometimes also called *base* objects.) Similarly, to disambiguate, we will talk about *primitives* instead of operations as far as low level objects are concerned. That is, a process invokes *operations* on a high level object and the implementation of these operations requires the process to invoke *primitives* of the underlying low level objects. When a a process invokes such a primitive, we say that the process performs a *step*.

The very notions of high level and low level are relative and depend on the actual implementations. An object might be considered high level in a given implementation and low level in another one. The object to be implemented is the high level one and the objects used in the implementation are the low level ones. In general, the intuition is that the low level objects might typically capture basic synchronization abstractions provided in hardware whereas the high level ones are those we want to emulate in software (what we call *implement*). Such emulations are strongly motivated by the desire to facilitate the programming of concurrent applications, i.e. to provide the programmer powerful synchronization abstractions encapsulated by high-level objects. Another motivation is to reuse programs initially devised with the high level object in mind in a system that does not provide such an object in hardware. Indeed, multiprocessor machines do not all provide the same basic synchronization abstractions. For instance, some modern machines provide $\mathsf{compare}\&\mathsf{swap}$ as a base object in hardware. Others do not and might provide instead $\mathsf{test}\&\mathsf{set}$ or simply some form of $\mathsf{registers}$. Providing an implementation of $\mathsf{compare}\&\mathsf{swap}$ using $\mathsf{test}\&\mathsf{set}$ would, for instance, make it possible to directly reuse, within an old machine, an application written for a modern machine.

An example is detailed in Figure 2 (Section 4.2) that describes the implementation of a FIFO queue (the high level object) from atomic low level objects denoted $NEXT$, $Q[1]$, $Q[2]$, etc. The low level object $NEXT$ provides the processes with the primitives $\mathsf{fetch}\&\mathsf{add}()$ and $\mathsf{read}()$, while each $Q[i]$ low level object provides the processes with the primitives $\mathsf{swap}()$ and $\mathsf{write}()$. So, a high level history is is made up of $Enq()$ and $Deq()$ operations, while a low level history is made up of invocations of the primitives $\mathsf{fetch}\&\mathsf{add}()$ and $\mathsf{read}()$ on the low level object $NEXT$, and $\mathsf{swap}()$ and $\mathsf{write}()$ on the low level objects $Q[x]$. This is schematically represented in the Figure 1.

Sometimes the low level objects are assumed to be atomic, and sometimes not. As shown in the second part of the book, it is sometimes useful to first implement intermediate objects that are not atomic, then implement the desired high level atomic objects on top of them.

2.2 Zooming into histories

When reasoning about atomicity, executions are represented with histories. As defined in the previous chapter, a history is a sequence of events, each representing an invocation or a reply on the high level shared objects to be implemented.

History of an implementation Reasoning about progress properties requires to zoom into the invocations and replies of the lower level objects on top of which the high level object is built. Hence, *lower level* histories are needed that depict events at the interface between the processes and the low level objects used in the implementation, i.e., the primitive events. Without such a zooming, it is not possible, for instance, to distinguish a faulty process that fails right after invoking a high level object operation, from one that keeps

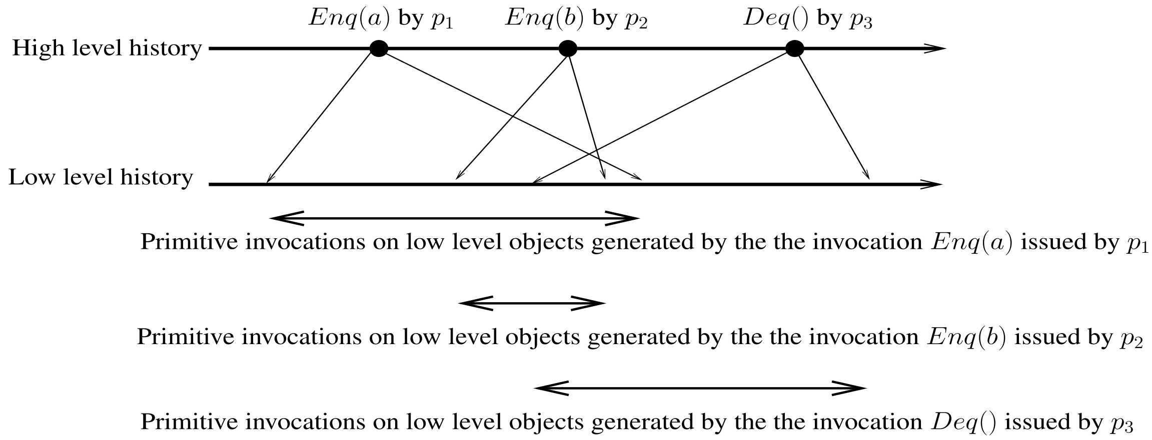

Figure 1: High level history low level history

executing the algorithm implementing that operation and invoking primitives of low level objects. We might want to require that the later obtains a matching reply and exempt the former from having to obtain a reply. So, when we talk about the *history of an implementation*, we implicit assume such a low level history, which is a refinement of the higher level history involving only the invocations and replies of the high level object.

When we consider the example of a FIFO high level object developed in Section 4.2, a high level history is a sequence built from the operations $Enq()$ and $Deq()$, while a low level history (or implementation history) is a sequence built from the primitives fetch&add() and read() associated with the $NEXT$ low level object, and the primitives swap() and write() associated with each $Q[i]$ low level object, that are generated from the invocations of the $Enq()$ and $Deq()$ operations.

The two faces of a process To better understand the very notion of a low level history, it is important to notice the two roles of a process. On the one hand, a process has the role of a *client* that sequentially invokes operations on the high level object and receives replies. On the other hand, the process has also the role of a *server* implementing the operations. To that end, it invokes primitives on lower level objects.

It might sometimes be convenient to think of the two roles of a process as executed by different entities. As a client, a process invokes object operations but does not control the way the low level primitives implementing these operations are executed. It even does not know how an object operation is implemented. Differently, as a server, a process executes the algorithm (made up of invocations of low level object primitives) associated with the high level object operation. Such an algorithm is typically described by an automaton (possibly with an unbounded number of states). The execution of a low level object primitive is called a *step*. A step constitutes an atomic unit of computation.

Scheduling and asynchrony The scheduling of steps in a system is specified by a *scheduler* (itself part of an operating system). This is outside of the control of processes and, in our context, it is convenient to think of a scheduler as an *adversary*. This is because, when devising a distributed algorithm, one has to cope with worst-case strategies of a scheduler that could defeat the algorithm.

A process is said to be *correct* when it executes an infinite number of steps, i.e., when the scheduler schedules an infinite steps of that process. (This "infinity" notion models the fact that the process executes as many steps as needed by the implementation). Otherwise, the process is said to be *faulty*. Sometimes it is convenient to see a faulty processes as a process that crashes and prematurely quits the computation. In the context of this book, we assume that processes do not deviate from the algorithm assigned to them. In other words, they are not malicious (we also say Byzantine).

Unless explicitly stated otherwise, the system is assumed to be *asynchronous* which means that the execution speed of each process is arbitrary, but positive until the process possibly crashes, in which case the speed becomes zero. It follows that there is no assumption on the processing speed of one process with respect to another. Basically, the way an asynchronous system progresses is controlled by a very weak scheduler, i.e., a scheduler whose only restriction lies in the fact that it cannot prevent forever a correct process from executing steps.

3 Progress properties

As pointed out, and strictly speaking, a trivial way to ensure atomicity would be to do nothing, i.e., not to return any response to any operation invocation. Besides this meaningless approach, a popular way to ensure atomicity is to use *critical sections* (say using *locks*) , preventing concurrent accesses to the same object. Every operation is executed as a critical section. Hence, when a process invokes an operation, it first requests a lock, and the algorithm of the operation is executed by the process only when the lock is acquired. If the lock is not available, the process waits until the lock is released. After a process obtains the result of an operation, it releases the corresponding lock.

As we discussed in the introduction of this book, such an implementation of a shared object has an inherent drawback: the progress of an operation issued by a process can be permanently prevented by the crash the process holding the lock. In practice, this might correspond to the situation where the process holding the lock is preempted for a long period, and all processes contending on the same object are blocked. When processes are asynchronous (i.e., the scheduler can arbitrarily preempt processes) which is the default assumption we consider, there is no way for a process to know whether another process has crashed (or was preempted for a long while) or is only very slow.

This book focuses on progress properties that preclude the use of critical sections and locks: they are instead strongly fault-tolerant in the sense that the crash of some processes does not prevent other processes from making progress. Several unbounded and bounded progress properties are presented below.

3.1 Solo, partial and global termination

- **Solo termination** An implementation satisfies the *solo termination* property (also called *obstruction-free* property) if any of its low level histories in which a single process p executes an infinite number of steps, provides a matching response to every high level operation invoked by p. (This property guarantees the progress of a process only when that process is the only correct process in the execution.)

  Intuitively, the aim of that property is to state that any process that executes *alone during long enough periods* always terminates its operations. As there is no way to say “long enough periods” in a time-free system such as an asynchronous system, the formal definition considers instead the existence of a single correct process. This formulation captures the fact that, after some finite time, the progress of that process can no longer be bothered by the progress of the other processes.

- **Partial termination** An implementation satisfies the *partial termination* property (also called *lock-free* or *non-blocking* property) if any suffix of its low level histories where at least one process invokes an infinite number of high level operations, has at least one process that obtains a matching response to each of its high level operation invocations.

  Intuitively, the partial termination property states that the object implementation is deadlock-free despite asynchrony and process crashes.

- **Global termination** An implementation satisfies *global termination* if, in any of its infinite low level histories, every process with an infinite number of steps has a matching response to each of its high level operation invocations. We will use also the term *wait-free* to characterize such implementations as it has been widely used in the literature. The term captures the very fact that every process terminates each of its operations without having to *wait* for steps of other processes.

  Intuitively, the wait-free property states that the object implementation is starvation-free despite asynchrony and process crashes.

Among the three guarantees defined above, solo termination is the weakest. It can be viewed as a form of wait-freedom restricted only to the case when there is a point in the computation (not necessarily known to the processes) after which there is no concurrency. As mentioned before, a shared object implementation is *obstruction-free* if each operation $op()$ terminates when the process invoking $op()$ runs alone for a sufficiently long period of time. When there are concurrent operations, these might or not terminate. This does not mean that obstruction-freedom is given for free. More precisely, an obstruction-free implementation of a concurrent object has to ensure that, despite concurrency, any operation that terminates does not provide incorrect results: the safety property associated with each operation has to always be ensured, independently of the concurrency context in which the operation is executed. In our context, this typically means that the atomicity of the high level object has to always be preserved.

Partial termination provides a higher level guarantee because the progress of at least one process is ensured, under all circumstances of concurrency. This is still weaker though than wait-freedom which requires progress of every process that keeps executing steps. Basically, wait-freedom stipulates that, in a finite number of its own steps, any process returns from each of its high level object invocations.

3.2 Bounded termination

Wait-freedom, the strongest among the liveness properties considered above, does not stipulate a bound on the number of steps that a process needs to execute before obtaining a matching response when it invokes a high level object operation. Typically, this number can depend on the behavior of the other processes. For example, it can be small if no other process performs any step, and increases when all processes perform steps (or the opposite), while remaining always finite despite up to $(n-1)$ process crashes.

- An implementation satisfies the **bounded wait-free** property if there is a bound B such that in any low level history (implementation history) every process p that invokes an operation receives a matching response within B of its own steps. (The B steps of p are not required to be consecutive.)

  In other words, there is no prefix of a low level history in which a process invokes an operation and executes B steps without obtaining a matching reply.

Showing that an implementation is bounded wait-free consists in exhibiting upper bound on the number of steps needed to return from any operation. That upper bound is usually defined by a function $f()$ on the number of processes (e.g., $O(n^2)$).

4 Atomicity and wait-freedom

Just as it is meaningless to ensure atomicity alone, without any progress guarantee, it is also meaningless to ensure any progress guarantee alone. Meaningful implementations are those that ensure both: ideal ones are those that ensure atomicity and wait-freedom.

Before diving into such implementations in the next chapters, it is important to ask whether every atomic object has an implementation that ensures wait-freedom and atomicity. In fact, it is easy to see that this would not be the case for objects with *partial* operations. By definition, the progress of such operations depend on concurrent invocations of other operations. That is, if an object's specification requires that a process does not return from an operation unless some other process executes some other operation concurrently, then it would be impossible to come up with even a solo-terminating implementation of this object, and this even if very powerful underlying base objects are used. As we discuss below however, an implementation that ensures wait-freedom and atomicity is always possible for objects with total operations.

4.1 Operation termination and atomicity

Besides *locality*, which we discussed in the previous chapter, a second noteworthy property provided by atomicity is on the termination of the high level object operations. In short, a pending invocation of a total operation is never required to wait for another operation to complete and yet preserves atomicity. This property has a fundamental consequence. It means that, per se, atomicity never forces a pending total operation to block. In other words, atomicity, per se, cannot prevent wait-freedom. Blocking (or even deadlock) can occur as an artifact of a particular implementation of atomicity, but is not inherent to atomicity. The following theorem captures this idea by stating that any (linearizable) history with a pending operation invocation can be extended with a response to that operation.

Theorem 1 *Let $inv[op(arg)]$ be the invocation event of a total operation that is pending in a linearizable history $\widehat{H}$. There exist a matching response event $resp[op(res)]$ such that the history $\widehat{H'} = \widehat{H}.resp[op(res)]$ is linearizable.*

Proof Let $\widehat{S}$ be a linearization of the partial history $\widehat{H}$. By definition of a linearization, $\widehat{S}$ has a matching response to every invocation. Assume first that $\widehat{S}$ includes a response event $resp[op(res)]$ matching the invocation event $inv[op(arg)]$. In this case, the theorem trivially follows as then $\widehat{S}$ is also a linearization of $\widehat{H'}$.

If $\widehat{S}$ does not include a matching response event, then $\widehat{S}$ does not include $inv[op(arg)]$ either. Because the operation $op()$ is total, there is a response event $resp[op(res)]$ matching the invocation event $inv[op(arg)]$ in every state of the shared object. Let $\widehat{S'}$ be the sequential history $\widehat{S}$ with the invocation event $inv[op(arg)]$ and a matching response event $resp[op(res)]$ added in that order at the end of $\widehat{S}$. $\widehat{S'}$ is trivially legal. It follows that $\widehat{S'}$ is a linearization of $\widehat{H'}$. $\Box_{Theorem\ 1}$

4.2 Example

To illustrate the inherent non-blocking feature of atomicity, and indirectly illustrate implementations that ensure atomicity and wait-freedom, consider a simple FIFO queue that can contain an unbounded number of items. The sequential specification of this object has been given in Section 2.2 of chapter 1.

The implementation example is voluntarily simple. (More sophisticated examples will be given in the next chapters of the book.) The system we consider here is made up of producers (clients) and consumers (servers) that cooperate through an unbounded FIFO queue. A producer process repeats forever the following two statements: it first computes a new item v, and then invokes the operation $Enq(v)$ to deposits v in the queue. Precisely because we assume that the queue is unbounded, the operation $Enq(v)$ is total.

Similarly, a consumer process repeats forever the following two statements: it first withdraws an item from the queue by invoking the operation $Deq()$, and then consumes that item. If the queue is empty, then the default value $\perp$ is returned to the invoking process. (This default value that cannot be deposited by a producer process.) Precisely because we do not preclude the possibility of returning $\perp$, the $Deq()$ operation also is total. We assume that no processing by the consumer is associated with the $\perp$ value.

The algorithm implementing the shared queue relies on an array $Q[0..\infty)$ used to store the items of the queue. Each entry of the array is initialized to $\perp$. Basically, the producer first locates the index of the next empty slot in the array Q, reserves it, and then stores the item in that slot. To dequeue a value, the consumer first determines the last entry of the array Q that has been reserved by a producer. Then, it scans the array Q in ascending order until it finds an item different from the default value $\perp$. If it finds one, it returns it. Otherwise, the default value is returned.

The algorithm is given in Figure 2. The *return* () statement terminates the operation (it does correspond to the response event associated with that operation). Lowercase letters are used for identifiers of local variables. Uppercase letters are used for shared variables. The implementation uses the following shared variables: $NEXT$ (initialized to 1) and the array Q, used to contain the values that have been produced and not yet consumed. The variable $NEXT$ is a pointer to the next slot of the array Q that can be used to deposit a new value. (This implementation could be optimized by reclaiming the slots from which items have been dequeued. But this is not the point here.)

The variable $NEXT$ is provided with two primitives denoted $\mathsf{read}()$ and $\mathsf{fetch\&add}()$. The invocation $NEXT.\mathsf{fetch\&add}(x)$ returns the value of $NEXT$ before the invocation and adds x to $NEXT$. Similarly, each entry $Q[i]$ of the the array is provided with two primitives denoted $\mathsf{write}()$ and $\mathsf{swap}()$. The invocation $Q[i].\mathsf{swap}(v)$ writes v in $Q[i]$ and returns the value of $Q[i]$ before the invocation.

The execution of the $\mathsf{read}()$, $\mathsf{write}()$, $\mathsf{fetch\&add}()$ and $\mathsf{swap}()$ primitives on the shared base objects ($NEXT$ and each variable $Q[i]$) are assumed to be atomic. They can be arbitrarily interleaved. Yet, everything appears as if no two such primitives are executed simultaneously. The primitives $\mathsf{read}()$ and $\mathsf{write}()$ are implicit in the code of Figure 2 (they are in the assignment statements denoted "$\leftarrow$").

There is no lock-based mechanism that forces parts of the code to be executed in mutual exclusion. Hence, no process can block forever other processes by crashing inside a critical section. Furthermore, each value deposited in the array by a producer will be withdrawn by a $\mathsf{swap}()$ operation issued by a consumer (assuming that at least one consumer is correct).

To better understand the algorithm, it is interesting to explore the two following situations.

- The first situation is when a producer crashes after $NEXT$ is increased by 1 and before the corresponding item is deposited in the array Q. That is, the producer reserves an index without ever using the corresponding slot. This conveys the fact that $NEXT$ represents an upper bound (and not a tight bound) on the number of items that are deposited in Q.) This can force consumers to explore more entries of Q than necessary. However, all these additional entries that are visited are equal to $\perp$. This means that, when a producer crashes in such a scenario, everything appears as if the producer had crashed before issuing the enqueue operation.

```
operation Enq(v):
    in ← NEXT.fetch&add (1);
    Q[in] ← v;
    return ()

operation Deq():
    last ← NEXT − 1;
    for i from 0 until last do
        aux ← Q[i].swap (⊥);
        if (aux ≠ ⊥) then return (aux) end_if
    end_do;
    return (⊥)
```

Figure 2: Enqueue and dequeue implementations

- The second situation is when during a dequeue operation, a consumer crashes right after having executed the statement $Q[i].\mathsf{swap}(aux)$. If $aux = \perp$, from an external observer point of view, everything appears as if the dequeue operation has never been issued. If $aux = v \neq \perp$, everything appears as if that value v has been obtained by the dequeue operation, the consumer crashing just after using it.

4.3 Correctness

We discuss first linearizability. Consider first the enqueue operation. Consider the case of an enqueue operation invoked by a process that crashes before any item is deposited in the array Q. Such operation appears as if it has never been invoked. Otherwise, the enqueue operation is linearized at the point when its item is deposited in the array Q. This point is unique: at most one process deposits an item in an index i of Q precisely because the index is reserved using a $\mathsf{fetch\&add}()$ primitive.

Consider a dequeue operation. If the operation returns some non-$\perp$ value, then the operation is linearized at the linearization point of the $\mathsf{swap}()$ primitive that returned that value. If the dequeue operation returns $\perp$, let t be the time after which the operation located an index i through $NEXT$. The dequeue operation is linearized after all the operations that dequeued the items available in Q at time t. Finally if the dequeue operation does not return, the process invoking it fails.

The implementation is clearly wait-free. No process can be blocked while executing an $enqueue()$ or $dequeue()$ operation, whatever the behavior of the other processes. A processes terminates an operation in a finite number of its own steps. The implementation is not bounded wait-free however. The $Enq()$ operation requires always three primitive invocations on the base objects, and is consequently bounded wait-free. Differently, the number of primitive invocations on base objects required by a $Deq()$ operation depends on the number of items that have been previously stored in the array Q. As this number cannot be bounded, the implementation is not bounded.

4.4 On the power of low level objects

The previous example shows that a FIFO queue, shared by an arbitrary number of processes, can be wait-free implemented from two types of base atomic objects, namely, an atomic object $NEXT$ whose type is defined by then pair of primitives $\mathsf{fetch\&add}()$ and $\mathsf{read}()$, as well as an array Q of atomic objects, the type of these objects being defined by the pair of primitives $\mathsf{write}()$ and $\mathsf{swap}()$.

This means that these base types are "powerful enough" to wait-free implement a FIFO queue shared by any number of processes. The investigation of the power of base object types to wait-free implement any concurrent object constitutes the topic addressed in the third part of the book.

4.5 Non-determinism

Before concluding this chapter, it is worthwhile to highlight the many sources of non-determinism in a concurrent system that need to be considered when devising a shared object implementation.

1. The scheduler of a concurrent system can orchestrate the steps of the processes in all kinds of ways and this is a source of non-determinism that any wait-free implementation has to cope with.

2. When modeling an actual execution of the system, we basically pick one possible history, among several possible ones, each with an underlying total order. This is another form of non-determinism. Indeed, some of the events of an execution might be simultaneous and each of their ordering leads to a specific history.

3. Finally, when seeking for a linearization of a concurrent history, we can also choose among several possible sequential histories. First, there might indeed be several ways of completing the original history, especially when non-deterministic objects are involved. Second, there might be several ways of ordering concurrent operations in the equivalent linearization.

5 Summary

This chapter has defined three progress properties: solo-termination, partial-termination and wait-freedom. A wait-free implementation of an atomic object is inherently robust in the following sense.

- It is inherently starvation-free. (This is an immediate consequence of the definition of the wait-free property.)

- It is $(n-1)$-resilient. This expresses the fact it naturally copes with up to $(n-1)$ process crashes.

Bibliographic notes

The notions of safety and liveness are due to Lamport [9]. A safety property (such as mutual exclusion or never deciding on conflicting values) stipulates that "nothing bad happens" and is consequently expressed as a property on all the prefixes of a computation. In contrast, a liveness property (such as eventually entering a critical section or eventually reaching agreement) stipulates that "something eventually happens". A liveness property involves consequently the whole computation. More on the liveness notion can be found in [1].

The notion of wait-freedom originates in the work of Lamport [8]. An associated theory has been developed by Herlihy [6].

The notion of solo-termination was presented implicitly in [3]. It has been introduced as a progress property in [7] under the name *obstruction-free* synchronization. That notion has been formalized in [2]. More developments on obstruction-freedom can be found in [4]. The minimal knowledge on process failures needed to go from obstruction-freedom to wait-freedom is investigated in [5].

References

[1] Alpern B. and Schneider F.B., Defining liveness. *Information Processing Letters*, 21(4):181-185, 18985.

[2] Attiya H., Guerraoui R. and Kouznetsov P., Computing with reads and writes in the absence of step contention. *Proc. 19rd Int'l Symposium on Distributed Computing (DISC'05)*, Springer-Verlag #3724, pp. 122-136, 2005.

[3] Fich F., Herlihy H. and Shavit N., On the space complexity of randomized synchronization. *Journal of the ACM*, 45(5):843-862. 1998.

[4] Fich F., Luchangco V., Moir M. and Shavit N., Obstruction-free algorithms can be practically wait-free. *Proc. 23rd Int'l Symposium on Distributed Computing (DISC'05)*, Springer-Verlag #3724, pp. 78-92, 2005.

[5] Guerraoui R., Kapalka M. and Kouznetsov P., The weakest failure detector to boost obstruction-freedom. *Proc. 20rd Int'l Symposium on Distributed Computing (DISC'06)*, Springer-Verlag #4167, pp. 399-412, 2006.

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

[7] Herlihy M., Luchangco V., Moir M., Obstruction-free synchronization: double-ended queues as an example. *Proc. 23rd IEEE Int'l Conference on Distributed Computing Systems I(CDCS '03)*, IEEE Computer Press, pp. 522-529, 2003.

[8] Lamport L., Concurrent reading and writing. *Communications of the ACM*, 20(11):806-811, 1977.

[9] Lamport. L., Proving the correctness of multiprocess programs. *IEEE Transaction on Software Engineering*, SE-3(2):125-143, 1977.