Fairness of Shared Objects
(Extended Abstract)

Michael Merritt* Gadi Taubenfeld**

Abstract. Fairness in concurrent systems can be viewed as an abstraction that bridges low-level timing guarantees and make them available to programmers with a minimal loss of power and a maximal ease of use. We investigate the implementation and power of a range of fairness models that are appropriate to the synchronous, semi-synchronous and asynchronous contexts of various concurrent systems.

1 Introduction

1.1 Motivation

Fairness is a powerful abstraction that has led to fruitful results in the theory of concurrent, distributed and nondeterministic programs. Various notions of fairness can be explained very briefly, but small differences have important consequences on the computability and complexity of concurrent systems. We investigate the implementation and power of a range of fairness models that are appropriate to the synchronous, semi-synchronous and asynchronous contexts of various concurrent systems.

While some previous work has effectively focused on designing algorithms using strong primitives, we try to lay out a more general framework. Consider for example the mutual exclusion problem. Almost all solutions to this problem assume a lower level "hardware" solution to the very problem they are solving. That is, the solutions are based upon an underline assumption of mutually exclusive atomic operations. Hence, as was pointed out by Lamport [17], they are not satisfactory for a fundamental study of this problem. The same can also be said about the fairness requirements. Almost all interprocess communication mechanisms require some underlying fairness in their implementation (usually, starvation-freedom), which may be the same fairness requirement of the problem they are being used to solve. This motivates the question of what can be done when the primitive objects are not assumed to satisfy the same fairness assumptions as the problem to be solved. We address such fairness issues here.

Unlike timing-based models and other fairness requirements which are defined with respect to processes (or schedulers), in this paper fairness is defined with respect to accesses made to distinct shared *objects*. In our framework it is possible

* AT&T Labs, 180 Park Av., Florham Park, NJ 07932-0971. mischu@research.att.com
** The Open University, 16 Klausner st., P.O.B. 39328, Tel-Aviv 61392, Israel. gadi@cs.openu.ac.il. Part of this work was performed while visiting at AT&T Labs.

to assume that not all shared objects satisfy that same fairness requirement, and that there are no fairness dependencies between objects.

The contribution of the paper is first in focusing on the fairness of the basic shared objects rather than on the fairness of the system as a whole. This gives rise to questions about the power of various objects under a variety of fairness assumptions. In this framework, we prove several new results, show how to express existing results, (including lower and upper bounds), and then generalize and strengthen them by changing the underlying fairness assumptions.

1.2 Definitions

We outline our model of distributed systems and define various notions of fairness. Processes are modeled as (possibly infinite) state-machines communicating via shared objects. A *configuration* of the system includes the state of each process and the state of each shared object. An *event* is either (1) an update of the internal state of a process, (2) a *request* by some process to access some shared object, (3) an update of the internal state of an object on behalf of a process, or (4) a *response* by a shared object to a pending request. We require an access' semantics (updating the object state and determining the value to return) to be the result of one or more internal steps of the object by the requesting process.

As in the standard interleaving semantics, a *run* α of the system is an alternating sequence $s_0 \xrightarrow{e_0} s_1 \xrightarrow{e_1} \cdots$ of configurations s_i and events e_i such that (1) the initial configuration s_0 satisfies appropriate initial conditions, and (2) every configuration s_{i+1} is derived from the previous configuration s_i by executing the event e_i.

We require the occurrence of each event to respect the associated process and/or object semantics. (In particular, we are not concerned with Byzantine failures.)

As noted, we are primarily interested in studying the impact of fairness properties imposed on accesses to shared objects. This study takes place under simple liveness properties imposed on processes: we assume any process that is not blocked by a non-terminating access either takes an infinite number of steps, or has failed (by terminating). Formally, a process p is *correct* in an execution α iff either p takes infinitely many steps in α or the last steps of p in α are of a pending request. (A request is pending if it is not followed by a response.) A process is *faulty* if it is not correct. [3]

We say that an algorithm or object implementation is *fault-free* if it meets its specification assuming all the processes are correct. Such a solution is *fully-resilient* if it works correctly regardless of the number of faulty processes.

Several of the fairness definitions below depend on the notion of a process trying to access an object. Let α be a run in which a process p issues a request

[3] We assume that all processes are infinite. That is, if an access is correct and if all its accesses all terminate, it takes an infinite number of steps. For terminating algorithms such as consensus, all but a finite number of those steps may be internal process events.

R to a shared object X. We say that p is *trying to access* X during the interval beginning with the last response to p preceding R from any shared object, or beginning with the initial state if no such response exists, ending with the response to R, or the remainder of α if the response does not exist. That is, we include in the interval in which we consider p to be trying to access X, the local computation by p preceding the explicit request R, and any subsequent steps of the run up to the response. This definition relates local computation time with the time needed to access shared objects. Most timing-based models assume that local computation time is negligible compared to the time it takes to access the shared memory. Although it is very convenient to assume that local computation takes zero time, in our framework it suffices to make this weaker assumption.

Some natural fairness properties for a set of shared objects, O, are defined below. (Each property implies the preceding.)

- *deadlock-freedom:* If a process is trying to access an object in O, then some process (not necessarily the same one) eventually succeeds in accessing some object in O.
- *starvation-freedom:* If a process is trying to access an object in O, then this process eventually succeeds. (Notice that one process can access objects in O arbitrarily many times ahead of another.)
- *bounded-waiting:* O is deadlock-free, and in any run there is a bound r such that whenever a process, p, is trying to access an object in O then no other process successfully accesses objects in O more than r times before p succeeds with its access.
- *r-bounded-waiting:* O is deadlock-free, and if a process is trying to access an object in O then it will succeed before any of the other processes is able to access O $r+1$ times.

We say that a given object X satisfies a fairness property F if the singleton set $\{X\}$ satisfies F, and call X an F object.

Simple Properties: Let O be a set of objects,

1. If each object in O is deadlock-free then O is deadlock-free. (The converse does not hold in general.)
2. Each object in O is starvation-free if and only if O is starvation-free.
3. If O is bounded-waiting then each object in O is bounded-waiting. (The converse does not hold in general.)
4. If O is r-bounded-waiting then each object in O is r-bounded-waiting. (The converse does not hold in general.)

1.3 Summary of results

We assume the reader is familiar with the definitions of mutual exclusion, consensus and leader election problems. A *decision* problem is, informally, a problem that can defined as an input-output relation. Thus, consensus and leader election are decision problems, while mutual exclusion is not. All statements below about mutual exclusion assume a fault-free model. For simplicity, read/write registers are called registers.

Deadlock-freedom: We prove the following,

- There is no deadlock-free mutual exclusion algorithm which uses only deadlock-free objects. [4]
- There is no fault-free solution to the consensus or leader election problems using deadlock-free registers. (There is a trivial fully-resilient solution to these problems using a single deadlock-free read-modify-write register.)
- Let P be a *decision* problem. There is a fully-resilient solution to P using deadlock-free objects of type X if and only if there is a fully-resilient solution to P using starvation-free objects of type X.
- Consensus is universal for deadlock-free objects.

The results show that deadlock-free objects are much weaker than starvation-free objects, and that both in the fault-free and the fault-tolerant models, different deadlock-free objects can have different computational power. The proof of the first result observes that deadlock-free objects are not sufficiently powerful to guarantee the transmission of even a single bit of information from a predefined sender process to a receiver process. The second result is proved by a reduction to known impossibility results for 1-resilient consensus [11, 19]. The proof of the third result relies on the simple structure of decision problems, while the fourth result is proved by a simple modification of the proof that consensus is universal for starvation-free objects [14].

Starvation-freedom: We make an observation which relates different fairness assumptions. It implies that in various cases, algorithms can be designed that perform well under fairness assumptions, but whose correctness (safety properties) are robust when those assumptions are violated.

- An algorithm A satisfies a safety property ϕ only if it does not matter whether the objects it uses are deadlock-free, starvation-free or bounded-waiting.

A similar type of result was proved in [1], showing that asynchronous systems and unknown-delay timing-based systems (see related work) are the same when safety properties are considered.

Bounded-waiting: We prove the following,

[4] There is a crucial distinction between a deadlock-free mutual exclusion *algorithm*, where some contending process must enter the critical section, and a deadlock-free mutual exclusion *object*. In the latter case, infinitely many accesses by contenders (accesses which return but deny the contenders the critical section) can satisfy the deadlock-free constraint, while the holder of the critical section hangs forever in its attempt to release it.

- Any deadlock-free mutual exclusion algorithm for n processes using a single bounded-waiting set of registers must use such a set which includes at least n registers.
- Any starvation-free mutual exclusion algorithm for n processes using a single bounded-waiting set of registers and test-and-set bits must use such a set which includes at least n instances of these objects.
- Using an unbounded number of bounded-waiting registers is a necessary and sufficient for construction of a fully-resilient consensus algorithm for an unspecified number of processes.
- Any fully-resilient consensus algorithm using a single bounded-waiting set of registers must use such a set which includes at least $\Omega(\sqrt{n})$ registers.
- Consensus is universal for bounded-waiting objects.

The first two results might suggest that there is not much difference between starvation-free objects and bounded-waiting objects, but the third and fourth results show that this is not the case. The proofs of the first and second results are based on proofs of corresponding results for starvation-free objects [8, 22]. A related lower bound of n starvation-free registers was proved in [21] for the unknown-delay timing-based model. To prove the sufficient condition of the third result, a fully-resilient consensus algorithm is presented. The algorithm is an adaptation of a consensus algorithm for the unknown-delay model using starvation-free registers [1]. The fourth result is also used to prove the necessary condition of the third result. Its proof is based on a construction from [10]. The last result is proved again by a simple modification to the proof in [14].

r-bounded-waiting: We prove that for any positive integer r:

- There is a deadlock-free mutual exclusion algorithm using a single r-bounded-waiting register.
- There is a fully-resilient consensus algorithm using a single r-bounded-waiting register.
- There is a starvation-free mutual exclusion algorithm, for an unspecified number of processes, which uses two r-bounded-waiting test-and-set bits and two starvation-free bits.
- There is a *fast* deadlock-free mutual exclusion algorithm which uses three r-bounded-waiting registers.
- There is a *fast* fully-resilient consensus algorithm which uses a single (3-valued) r-bounded-waiting register, and two deadlock-free bits.

These results demonstrate that, no matter how big r is, r-bounded-waiting objects are much stronger than bounded-waiting objects. The mutual exclusion and consensus algorithms which use a single r-bounded-waiting register are based on a simple timing-based algorithm due to Fischer [18]. The algorithm which is used to prove the third result is based on an algorithm due to Friedberg and Peterson [13], which originally used weak semaphores. (Recall that achieving constant space is not possible using bounded-waiting registers and bounded-waiting test-and-set bits.) The two algorithms which are used to prove the fourth and fifth results are adaptations of timing-based algorithms from [4].

1.4 Related Work

The text [12] brings together most of the known approaches for proving properties of programs and, in particular, focuses on proving the termination of programs under various fairness assumptions about the processes' behavior (such as unconditional, weak and strong fairness). Our focus in this paper is less on formal verification issues and more on algorithmic issues that arise due to various fairness assumptions on how pending requests are served by shared objects.

In [3, 4, 18, 21, 23], a shared memory model, called the *known-delay* model (also called delay-based), is considered where it is assumed that there is an upper bound on the time required for a single access to shared memory, and that the bound is known to all the processes. This is a powerful model that enables efficient algorithms for various problems, and is similar to a system where *all* shared objects are assumed to be included in one big set which is r-bounded-waiting for some known r. Some of the work for the above timing-based model has been extended to the *unknown-delay* model, where an upper bound exists on the memory-access time, but is not known *a priori* [1]. Such a model is stronger than the asynchronous model, enabling the design of algorithms for problems that are unsolvable in the asynchronous model.

The notion of *finitary fairness* was introduced in [2]. Finitary fairness requires that for every run of a system there is an unknown bound k such that no enabled transition is postponed more than k consecutive times. It is shown in [2], that under finitary fairness formal verification can be simplified and that fully-resilient consensus is solvable (since finitary fairness can replace the timing assumption for the consensus algorithm in [1]). Bounded fairness was also considered in [16], by introducing a new operator into temporal logic. Both systems that satisfy the finitary fairness assumption and unknown-delay timing-based systems are basically the same as systems where *all* shared objects are considered to be one big bounded-waiting set.

The "Archimedean" systems explored in [25, 24] provide an interesting concept for using fairness as abstraction in message passing systems, and has motivated some of the work for shared memory systems mentioned above. In such systems, relative but unknown bounds are assumed on the speeds of network components, and efficient algorithms can be designed that exploit this synchrony assumption. Models slightly stronger than Archimedean systems are studied in [6, 5, 9, 15].

Two common notions used in the literature are those of *non-blocking* and *wait-free* objects [14]. The first corresponds to implementations which are both fully-resilient and deadlock-free, while the second corresponds to implementations which are both fully-resilient and starvation-free. (While wait-freedom requires only that every operation (process) always terminates within a finite number of steps, bounded wait-freedom requires that it terminates within a fixed and pre-determined number of steps [7].) An interesting notion of fairness, called closed schedulers, is explored in [20].

2 Deadlock-freedom

Assuming the fairness assumption is deadlock-freedom, several results are presented. The first result implies that deadlock-free objects are much weaker than starvation-free objects.

Theorem 1. *There is no deadlock-free mutual exclusion algorithm which uses only deadlock-free objects.*

Proof. The proof is based on the observation that deadlock-free objects are not sufficiently powerful to guarantee the transmission of even a single bit of information from a predefined sender process to a receiver process. (Suppose we want to block transmission from process P to Q. Run Q in isolation. There is a deadlock-free run of P and Q in which Q takes the same steps: at any object o touched by both P and Q one of two cases holds. Either Q accesses o a finite number of times, all before P's first access to o, or Q accesses o an infinite number of times, preventing P's first access from terminating.) Consider a mutual exclusion system in which process P holds the critical section and accesses shared variables to release it forever: by this observation, there is a deadlock-free run in which P never succeeds in informing Q that the critical section is available, so Q can never enter it. □

There are many deadlock-free mutual exclusion algorithms using various starvation-free objects, such as, starvation-free registers, test&set, read-modify-write objects. Thus, such objects cannot be implemented using deadlock-free objects, even in a fault-free model.

The next result, together with the observation that there is a trivial, fully-resilient solution to consensus and leader election using a single deadlock-free read-modify-write register, shows that, even in a fault-free model, deadlock-free objects have different computational power. (The trivial solution is that the first process to access the read-modify-write register writes its proposed value, and every other process reads this value and decides on it. Since every correct process stops accessing the object after a single access, in deadlock-free runs every correct process must eventually finish its access.)

Theorem 2. *There is no fault-free solution to the consensus or leader election problems using deadlock-free registers.*

The proof is a reduction to the impossibility of 1-resilient consensus using starvation-free (read/write) registers [11, 19].

Our next theorem shows that, unlike the result of Theorem 1, in the context of decision problems (consensus, leader election, etc.) deadlock-free and starvation-free fairness are interchangeable.

Theorem 3. *Let P be a* decision *problem. There is a fully-resilient solution to P using deadlock-free objects of type X if and only if there is a fully-resilient solution to P using starvation-free objects of type X.*

(At first glance, using Herlihy's result that consensus is universal [14], Theorem 3 seems to contradict Theorem 1 by implying that deadlock-free objects has the same computational power as starvation-free objects. This problem is resolved by observing that Herlihy's construction makes use of starvation-free registers, and hence holds only for starvation-free objects.)

We say that an object of type T is *universal for deadlock-free objects*, if there is a fully-resilient implementation of any (linearizable) deadlock-free object using any number of objects of type T together with deadlock-free registers. The next result is that any deadlock-free object which can implement consensus for n processes is universal for deadlock-free objects in a system of n (or fewer) processes.

Theorem 4. *Consensus is universal for deadlock-free objects.*

The proof is constructed by replacing starvation-free registers with deadlock-free registers in Herlihy's proof that consensus is universal for starvation-free objects [14].

3 Starvation-freedom

Most algorithms, lower bounds, and impossibility results for the shared memory model assume that the primitive shared objects are starvation-free. We make an observation which relates different fairness assumptions, showing that using deadlock-free, starvation-free or bounded-waiting objects is the same as far as safety properties are concerned. (A property ϕ is a *safety* property if the following holds: an infinite run satisfies ϕ if and only if all finite prefixes of the run satisfy ϕ, that is, a safety property has to be *prefix-closed.*)

Theorem 5. *An algorithm A satisfies a safety property ϕ only if it does not matter whether the objects it uses are deadlock-free, starvation-free or bounded-waiting.*

Proof. To show that it does not matter whether each of the objects being used is deadlock-free or starvation-free, it is enough to observe that the set of (legal) finite runs of A is the same whether each of the objects is deadlock-free or starvation-free. If A satisfies a safety property ϕ with starvation-free objects then, clearly, it satisfies ϕ with bounded objects. To prove the other direction, suppose A does not satisfy ϕ with starvation-free objects. If ϕ is a safety property then there is a finite run which violates ϕ. This run is also possible when using bounded-waiting objects since we can chose the bound to be as big as needed. Thus A does not satisfy ϕ also with bounded objects. □

4 Bounded-waiting

Of all the fairness assumptions covered in this papers, bounded-waiting is probably the most useful one. While bounded-waiting objects abstract from implementation details, they are arguably a better approximation of real shared objects compared to the (usually assumed) starvation-free objects. Furthermore, they enable us to design algorithms for problems that are unsolvable using starvation-free objects.

In the case of starvation-free registers, any deadlock-free mutual exclusion algorithm for n processes must use at least n such registers [8]. Similar result holds also for bounded-waiting registers.

Theorem 6. *Any deadlock-free mutual exclusion algorithm for n processes using a single bounded-waiting set of registers must use such a set which includes at least n registers.*

Proof. To prove the corresponding results for starvation-free registers [8], Burns and Lynch have shown how to construct a finite run which violates mutual exclusion, for any algorithm which uses less than n starvation-free registers. Since any such finite run is a bounded-waiting run (for some bound r), this construction similarly constrains the number of bounded-waiting registers needed. □

The lower bound of a bounded-waiting set of n registers is tight since there exists a deadlock-free mutual exclusion algorithm which uses only n starvation-free bits [8]. While n bounded-waiting registers are needed, the problem can be solved with only one r-bounded-waiting register, no matter how big r is. A related lower bound of n (starvation-free) registers was proved in [21] for the unknown-delay timing-based model.

In the case of starvation-free registers and starvation-free test-and-set bits, any starvation-free mutual exclusion algorithm for n processes must use at least n such objects [22]. Our next result shows that a similar result holds for the bounded-waiting case.

Theorem 7. *Any starvation-free mutual exclusion algorithm for n processes using a single bounded-waiting set of registers and test-and-set bits must use such a set which includes at least n instances of these objects.*

Proof. To prove the result for starvation-free objects [22], Peterson constructs a finite run which violates mutual exclusion. As above, this construction also holds for a set of bounded-waiting registers and test-and-set bits. □

We will see below that the problem can be solved with only a constant number of r-bounded-waiting registers and test-and-set bits, no matter how big r is.

While the last two results seem to suggest that there is not much difference between starvation-free objects and bounded-waiting objects, the next result shows that this is not the case. Recall, that consensus cannot be solved using starvation-free registers only, even in the presence of a single faulty process [11, 19].

Theorem 8. *An unbounded number of bounded-waiting registers is a necessary and sufficient for constructing a fully-resilient consensus algorithm for an unspecified number of processes.*

The sufficient condition is proved by presenting a fully-resilient consensus algorithm using an unbounded number of bounded-waiting registers.

Theorem 9. *There is a consensus algorithm for an unspecified number of processes using an unbounded number of bounded-waiting registers.*

Proof. The algorithm below is an adaptation of the algorithm for the unknown-delay model from [1]. The algorithm is *fast*: in the absence of contention, a process needs to take only a constant number of steps before terminating.

CONSENSUS ALGORITHM: `program for process` i `with input` in_i.
out: starvation-free register, initially $\perp$
$x[1..\infty, 0..1]$ array of starvation-free registers, initially 0
$y[1..\infty]$ array of bounded-waiting registers, initially $\perp$
$round$, v: local registers initially 1 and in_i, resp.

```
while out = ⊥ do
    x[round, v] := 1;
    if y[round] = ⊥ then y[round] := v fi;
    if x[round, v̄] = 0 then out := v
                 else for j = 1 to round do dummy := y[round] od;
                      v := y[round];
                      round := round + 1 fi
od;
decide(out).
```

Due to lack of space, we only give a brief explanation. The algorithm proceeds in rounds. Each process has a preference for the decision value in each round; initially this preference is the input value of the process. We assumed that all the infinite number of bounded-waiting registers $y[i]$ are r-bounded-waiting for some unknown number r. Once the algorithm reaches round r, it is guaranteed that all processes will move to the next round with the same preference. Once all processes in a round have the same preference, then a decision is reached in that round.[5] □

The next result is a space lower bound for solving consensus, and is also used to prove the necessary condition of Theorem 8.

Theorem 10. *Any fully-resilient consensus algorithm using a single bounded-waiting set of registers must use such a set which includes at least $\Omega(\sqrt{n})$ registers (even if each register has unbounded size).*

[5] We can make the algorithm more time efficient by modifying the for-loop as follows: **for** $j = 1$ **to** $round!$ **do** $dummy = y[round]$ **od**; Let r denotes the unknown bound of the bounded-waiting registers $y[i]$. It can be shown that the step complexity of the algorithm (with r factorial) is $O(r\frac{\log r}{\log \log r})$, and that this bound is tight.

Proof. In [10] Fich, Herlihy and Shavit show that when less than $c\sqrt{n}$ starvation-free registers are used for some sufficiently small constant c, there is a *finite* run in which processes decide on opposite values. This is then used to prove a $\Omega(\sqrt{n})$ lower bound for randomized consensus. This construction holds also assuming the registers are bounded-waiting, since the finite run is a bounded-waiting run. □

Extensions of the cited theorem noted in [10] also apply in this context: hence a similar bound to that in Theorem 10 holds on the number of swap objects or history-less objects needed to solve consensus. As we noted previously, there are several deadlock-free objects, such as read-modify-write or compare-and-swap, such that consensus can be solved using only one such object. Thus, any implementation of such objects using only swaps and registers requires $\Omega(\sqrt{n})$ swaps and registers.

In a related result [2], it is proved that under finitary fairness, there is no fully-resilient consensus algorithm even for two processes that uses a fixed number of bounded-size registers. This result and Theorem 10 still leave open the question of tight bounds on the number of bounded-waiting registers (some of which must be of unbounded-size) necessary to solve the consensus problem as a function of the number of processes.

An object is *universal for bounded-waiting objects*, if together with bounded-waiting registers it can implement any other bounded object (assuming any number of faults may occur). The next result is that any bounded-waiting object which can implement consensus for n processes is universal for bounded-waiting objects in a system of n (or fewer) processes.

Theorem 11. *Consensus is universal for bounded-waiting objects.*

The proof is constructed by replacing starvation-free registers with bounded-waiting registers in Herlihy's proof that consensus is universal for starvation-free objects [14].

5 r-bounded-waiting

In these section we present several algorithms which make use of r-bounded-waiting objects. The algorithms demonstrate that r-bounded-waiting objects are much stronger than bounded-waiting objects. We start with a very simple observation about r-bounded-waiting registers.

Theorem 12. *For any positive integer r, there is a deadlock-free mutual exclusion algorithm and a fully-resilient consensus algorithm each uses a single r-bounded-waiting register.*

Proof. The simple mutual exclusion algorithm uses a single r-bounded-waiting register and is based on a timing-based algorithm due to Fischer [18]. The register, called x, is initially 0. A process first waits until $x = 0$ and then assigns its *id* to x. Then, it reads x exactly r times. The fact that x is an r-bounded-waiting register ensures that after a process finishes reading, the value of x remains unchanged until some process leaving its critical section sets x back to 0. If x hasn't been changed and hence equals the process id, the process can safely enter its critical section, otherwise it goes to the beginning of its code and repeats this procedure. A consensus algorithm is constructed in a similar way. □

Theorem 13. *For any positive integer r, there is a starvation-free mutual exclusion algorithm, for an unspecified number of processes, which uses two r-bounded-waiting test-and-set bits and two starvation-free bits.*

Proof. The algorithm presented is based on an algorithm due to Friedberg and Peterson [13], which originally used weak semaphores. Recall that achieving constant space is not possible using bounded-waiting registers and bounded-waiting test-and-set bits (Theorem 7). For luck of space, the algorithm is given without additional explanation. In the code that follows we use **await** *test-and-set*(*lock*) $=$ 0 with the meaning **while** *test-and-set*(*lock*) $\neq 0$ **do** skip **od**.

THE ALGORITHM: `program of a process.`

T_0 and T_1: r-bounded-waiting test-and-set bits, both are initially 0;
queue, empty: starvation-free bits initially 0 and *false*, respectively;
myqueue, otherqueue: local bits;
counter: local register;

```
myqueue := queue;                           %remember where to wait
await test-and-set(T_myqueue) = 0           %wait to be flushed
if queue = myqueue                          %first to enter?
then                                        %become the doorkeeper
    otherqueue := myqueue + 1 (mod 2);
    await test-and-set(T_otherqueue) = 0;   %take the other bit
    queue := otherqueue;                    %change queue
    repeat                                  %flush out waiting processes
        if empty := false then counter := 0
        else counter := counter + 1 fi;
        empty := true;
        release(T_myqueue);
        await test-and-set(T_myqueue) = 0;
    until counter = r;                      % r-boundedness is used
    Critical Section;
    release(T_otherqueue);                  %give up the bit
else                                        %not a doorkeeper
    empty := false;
    Critical Section;
fi;
release(T_myqueue);                         %give up the original bit
```

The algorithm satisfies mutual exclusion deadlock freedom, and 2-bounded waiting (i.e., a process can enter its critical section ahead of another only twice). □

A mutual exclusion algorithm is *fast* if in the absence of contention, a process can always enter its critical section and exit it in a constant number of steps. I.e., the number of steps in this case does not depend on r. Similarly, a consensus algorithm is *fast* if, in absence of contention, a process can decide after a constant number of steps.

Theorem 14. *For any positive integer r, there is a* fast *deadlock-free mutual exclusion algorithm which uses three r-bounded-waiting registers.*

Proof. A solution is presented in which, in the absence of contention, a process can always enter its critical section and exit it in a constant number of steps. The solution is an adaptation of a known timing-based algorithm from [4]. For luck of space, the code of the algorithm is given without additional explanation.

A FAST ALGORITHM: `process` i`'s program.`
x, y, z: r-bounded-waiting registers, initially $y = 0$ and $z = 0$.

start: $x := i$;
    await $(y = 0)$;
    $y := i$;
    if $x \neq i$ **then for** $i = 1$ **to** $2r$ **do**
            dummy $:= x$; *dummy* $:= y$; *dummy* $:= z$ **od**;
            if $y \neq i$ **then goto** *start* **fi**;
            await $(z = 0)$
        else $z := 1$ **fi**;
    critical section;
    $z := 0$;
    if $y = i$ **then** $y := 0$ **fi**;

The algorithm satisfies mutual exclusion and deadlock-free. □

Theorem 15. *For any positive integer r, there is a* fast *fully-resilient consensus algorithm which uses a single (3-valued) r-bounded-waiting register, and two deadlock-free bits.*

Proof. The simple fast algorithm presented below is an adaptation of a known timing-based algorithm from [4].

FAST CONSENSUS: `program for process` i `with input` in_i`.`
$x[0]$ and $x[1]$: deadlock-free bits, initially 0
y: r-bounded-waiting register, initially $\perp$

$x[in_i] := 1$;
if $y = \perp$ **then** $y := in_i$ **fi**;
if $x[1 - in_i] = 1$ **then for** $i = 1$ **to** r **do** *dummy* $:= y$ **od fi**;
decide(y).

□

Finally, we point out that using r-bounded-waiting registers it is simple to implement r'-bounded-waiting test-and-set bits (with reset) and r'-bounded-waiting compare-and-swap objects, for some r'. These implementations are similar to the corresponding timing-based constructions from [3, 23].

6 Discussion

We have defined fairness with respect to objects. It is possible, however, to define fairness with respect to operations. For example, consider registers which support deadlock-free read operations and starvation-free write operations. With such registers it is possible to solve deadlock-free mutual exclusion and hence such registers are strictly stronger than deadlock-free read/write registers. Also, all our fairness definitions can be generalized to apply also to sets of sets etc. of objects. Let O be the set $\{\{X_1, X_2\},\{X_3, X_4\}\}$. If X_1, X_2, X_3, X_4 are all bounded-waiting objects then O is starvation-free (but not bounded-waiting).

In the way object fairness is defined, an object bestows the same fairness guarantees to all participating processes. This can be generalize by defining fairness w.r.t. a pair of a process and an object. This would enable the same object to guarantee starvation-freedom to one process and bounded-waiting to another.

There are several other fairness assumptions that can be further explored. For example, our definitions can be generalized to reflect priorities of processes. Yet another interesting direction is to define a weaker version of r-bounded-waiting fairness, by assuming that the r-bounded-waiting requirement may not hold from the beginning of each run, but is guaranteed to hold *eventually*.

References

1. R. Alur, H. Attiya, and G. Taubenfeld. Time-adaptive algorithms for synchronization. *SIAM Journal on Computing*, 26(2):539–556, April 1997.
2. R. Alur and T. Henzinger. Finitary fairness. In *Proc. 9th IEEE Symp. on Logic in Computer Science*, pages 52–61, 1994.
3. R. Alur and G. Taubenfeld. How to share an object: A fast timing-based solution. In *Proceedings of the 5th IEEE Symposium on Parallel and Distributed Processing*, pages 470–477, December 1993.
4. R. Alur and G. Taubenfeld. Fast timing-based algorithms. *Distributed Computing*, 10:1–10, 1996.
5. H. Attiya and T. Djerassi-Shintel. Time bounds for decision problems in the presence of timing uncertainty and failures. *Lecture Notes in Computer Science*, 725, 1993.
6. H. Attiya, C. Dwork, N. Lynch, and L. Stockmeyer. Bounds on the time to reach agreement in the presence of timing uncertainty. In *Proc. 23rd ACM Symp. on Theory of Computing*, pages 359–369, May 1991.
7. H. Brit and S. Moran. Wait-freeom vs. bounded wait-freeom in public data structures. In *Proc. 13th ACM Symp. on Principles of Distributed Computing*, pages 52–60, August 1994.

8. J. N. Burns and N. A. Lynch. Bounds on shared-memory for mutual exclusion. *Information and Computation*, 107(2):171–184, December 1993. (Also, in Proc. of 18th Annual Allerton Conference on Communication, Control and Computing, 1980, pages 833–842.).
9. C. Dwork, N. Lynch, and L. Stockmeyer. Consensus in the presence of partial synchrony. *Journal of the ACM*, 35(2):288–323, 1988.
10. F. Fich, M. Herlihy, and N. Shavit. On the space complexity of randomized synchronization. In *Proc. 12th ACM Symp. on Principles of Distributed Computing*, pages 241–250, August 1993.
11. M. J. Fischer, N. A. Lynch, and M. S. Paterson. Impossibility of distributed consensus with one faulty process. *Journal of the ACM*, 32(2):374–382, April 1985.
12. N. Francez. *Fairness*. Springer-Verlag, 1986.
13. S. A. Friedberg and G. L. Peterson. An efficient solution to the mutual exclusion problem using weak semaphores. *Information Processing Letters*, 25(5):343–347, 1987.
14. M. Herlihy. Wait-free synchronization. *ACM Trans. on Programming Languages and Systems*, 11(1):124–149, January 1991.
15. A. Herzberg and S. Kutten. Efficient detection of message forwarding faults. In *Proc. 8th ACM Symp. on Principles of Distributed Computing*, pages 339–353, 1989.
16. D. N. Jayasimha and N. Dershowitz. Bounded fairness. Technical Report 615, Center for Supercomputing Research and Development, University of Illinois, Urbana, IL, December 1986.
17. L. Lamport. The mutual exclusion problem: Part I – a theory of interprocess communication. *Journal of the ACM*, 33:313–326, 1986.
18. L. Lamport. A fast mutual exclusion algorithm. *ACM Trans. on Computer Systems*, 5(1):1–11, 1987.
19. M. C. Loui and H. Abu-Amara. Memory requirements for agreement among unreliable asynchronous processes. *Advances in Computing Research*, 4:163–183, 1987.
20. R. Lubitch and S. Moran. Closed schedulers: a novel technique for analyzing asynchronous protocols. *Distributed Computing*, 8(4):203–210, 1995.
21. N. Lynch and N. Shavit. Timing-based mutual exclusion. In *Proceedings of the 13th IEEE Real-Time Systems Symposium*, pages 2–11, December 1992.
22. G. L. Peterson. New bounds on mutual exclusion problems. Technical Report TR68, University of Rochester, February 1980 (Corrected, Nov. 1994).
23. S. Ramamurthy, M. Moir, and J. H. Anderson. Real-time object sharing with minimal system support (extended abstract). In *Proceedings of the 15th Annual ACM Symposium on Principles of Distributed Computing*, pages 233–242, May 1996.
24. Paul G. Spirakis and Basil Tampakas. Efficient distributed algorithms by using the Archimedean time assumption. In *5th Annual Symposium on Theoretical Aspects of Computer Science*, volume 294 of *lncs*, pages 248–263. Springer, 1988.
25. Paul M. B. Vitányi. Distributed elections in an Archimedean ring of processors (preliminary version). In *Proceedings of the Sixteenth Annual ACM Symposium on Theory of Computing*, pages 542–547, 1984.

This article was processed using the LaTeX macro package with LLNCS style