**Noname manuscript No.**
(will be inserted by the editor)

# Waiting in Concurrent Algorithms

**Gadi Taubenfeld**

Received: date / Accepted: date

**Abstract** Between the two extremes, *lock-based* algorithms, which involve "a lot of waiting", and *wait-free* algorithms, which are "free of locking and waiting", there is an interesting spectrum of different levels of waiting. This unexplored spectrum is formally defined and its properties are investigated. New progress conditions, called $k$-waiting, for $k \geq 0$, which are intended to capture the "amount of waiting" of processes in asynchronous concurrent algorithms, are introduced. To illustrate the utility of the new conditions, they are used to derive new lower and upper bounds, and impossibility results for well-known basic problems such as consensus, election, renaming and mutual exclusion. Furthermore, the relation between waiting and fairness is explored.

**Keywords** Synchronization · wait-freedom · locks · enabled process · enabling step · $k$-waiting · consensus · election · renaming · mutual exclusion.

## 1 Introduction

Concurrent access to a data structure shared among several processes must be synchronized in order to avoid interference between conflicting operations. Mutual exclusion locks are the de facto mechanism for concurrency control on concurrent data structures: a process accesses the data structure only inside a critical section code, within which the process is guaranteed exclusive access. However, using locks may degrade the performance of synchronized concurrent applications, as it enforces processes to wait for a lock to be released.

A preliminary version of the results presented in this paper, appeared in *4th international conference on networked systems (NETYS 2016)*, Marrakech, Morocco, May 2016 [25].

The Interdisciplinary Center
P.O.Box 167, Herzliya 46150, Israel
E-mail: tgadi@idc.ac.il

A promising approach, which overcomes some of these difficulties, is the design of concurrent data structures and algorithms which avoid locking. The advantages of such algorithms are that they are not subject to priority inversion, are resilient to failures, and do not suffer significant performance degradation from scheduling preemption, page faults or cache misses. Although desirable, such implementations are often complex, difficult to design, inefficient, memory consuming and require the use of strong synchronization primitives.

Implementations which use locks are usually easier to program than implementations which avoid locking and waiting. Such lock-based implementations usually require "a lot of waiting", compared to implementations which avoid waiting, and may force operations that do not conflict to wait for one another, precluding disjoint-access parallelism.

In this paper, we show that between these two extremes: "a lot of waiting" (i.e., locks) and "free of locking and waiting", there is an interesting spectrum of different levels of waiting. We identify and formally define this unexplored spectrum, by introducing new progress conditions, called $k$-waiting, for $k \geq 0$, which are intended to capture the "amount of waiting" of processes in asynchronous concurrent algorithms.

Intuitively, these new progress conditions can be described as follows. A process is *enabled*, if it does not need to wait for an action by any other process in order to complete its operation. A step is an *enabling* step, if after executing that step at least one process which was not enabled becomes enabled. For a given $k \geq 0$, the *$k$-waiting* progress condition guarantees that every process that has a pending operation, will always become enabled once at most $k$ enabling steps have been executed.

To illustrate the utility of the new progress conditions, we use them to derive new lower and upper bounds, and impossibility results for well-known basic problems such as consensus, election, renaming and mutual exclusion. Furthermore, the relation between waiting and fairness is explored.

## 2 The $k$-waiting progress conditions

In this section, we discuss and formally define the new notion of $k$-waiting. An implementation of an operation may involve several basic steps. A basic step, like reading, updating or testing, may involve accessing a shared memory location. An implementation of each operation of a concurrent data structure is divided into two continuous sections of code: the *doorway* code and the *body* code. When a process invokes an operation it first executes the doorway code and then executes the body code. The *doorway*, by definition, must be *wait-free*: its execution requires only bounded number of steps and hence always terminates.

A process executes a sequence of steps as defined by its algorithm. A (global) *state* of an algorithm is completely described by the values of the (local and shared) objects and the values of the location counters of all the processes. An execution (also called a run) is defined as a sequence of alter-

nating states and steps (also referred to as events). It is convenient to define an execution as a sequence of steps omitting all the states except the initial state. Since the states in an execution are uniquely determined by the steps and the initial state, no information is lost by omitting the states. For two executions $r$ and $r'$, $r'$ is an *extension* of $r$ if and only if $r$ is a prefix of $r'$.

A *beginning* process is a process that is about to start executing the first step of some operation. An *active* process, is a process that has already executed the first step of some operation, but has not completed that operation yet. A process has *passed* the doorway of a given operation, if it has finished the doorway code and reached the body code of that operation.

### 2.1 An enabled process and an enabling step

The following definitions refer to both beginning and active processes.

> **A strongly enabled process:** A process is *strongly enabled* at the end of a given execution $r$, if, at the end of any possible extension of $r$, it does not need to wait for an action by any other process in order to complete its operation, nor can an action by any other process prevent it from doing so. Thus, by executing sufficiently many steps, it will be able to complete its operation, independently of the actions of the other processes.[1]

Being strongly enabled is a *stable* property, if a process is strongly enabled at some point then, by definition, it must also be strongly enabled at any later point during the operation.

> **A weakly enabled process:** A process is *weakly enabled* at the end of a given execution $r$, if, at the end of any possible extension of $r$, it does not need to wait for an action by any other process in order to complete its operation, however, actions by other processes (while they occur) may prevent it from doing so. Thus, at the end of *any* extension of $r$, by executing sufficiently many steps, the process will be able to complete its operation, provided that (from some point on) other processes do not take steps. Put another way, process $p$ is *weakly enabled* at the end of a given execution $r$, if at the end of any possible extension of $r$, when $p$ runs alone it eventually terminates.

We say that a process is *disabled* if it is not weakly enabled. We notice that once a process becomes weakly enabled, it cannot later become disabled. If a process is weakly enabled at some point then, by definition, it must be weakly or strongly enabled at any later point during the operation. Thus being weakly enabled is also a stable property. A strongly enabled process is, by definition, also weakly enabled. For an execution (run) $r$ and a step $s$, we denote by $r; s$ the execution obtained by extending $r$ with the step $s$.

[1] In the case of a beginning process, "its operation" means the operation that the process is about to start executing.

> **An enabling step:** A step is a *strong enabling* (resp. *weak enabling*) step, at the end of a given execution, if after executing that step at least one process which was not strongly (resp. weakly) enabled becomes strongly (resp. weakly) enabled. More formally, $s$ is a strong (resp. weak) enabling step at the end of execution $r$, if there exists at least one process, say $p$, such that $p$ is not strongly (resp. weakly) enabled at the end of $r$ and $p$ is strongly (resp. weakly) enabled at the end of $r; s$.

We notice that a strong enabling step is not necessarily also a weak enabling step, and vice versa. A single strong (resp. weak) enabling step may cause several processes, not necessarily just one, to become strongly (resp. weakly) enabled. If $s$ is a strong (resp. weak) enabling step *at* the end of $r$, and $r'$ is an extension of $r; s$, then we say that $s$ is a strongly (resp. weakly) enabling step *in* $r'$. For two executions $r$ and $r'$, we use the notation $r \leq r'$ to denote the fact that $r'$ is an extension of $r$. When $r \leq r'$, we denote by $(r' - r)$ the suffix of $r'$ obtained by removing $r$ from $r'$.

### 2.2 $k$-waiting

The following definition is central for our investigation.

> **$k$-waiting:** For $k \geq 0$, the strong (resp. weak) *$k$-waiting* progress condition guarantees that *every* process, that has *passed* its doorway, will always become strongly (resp. weakly) enabled once at most $k$ strong (resp. weak) enabling steps have been executed. More formally, the strong (resp. weak) *$k$-waiting* progress condition guarantees that, for every two executions $r$ and $r'$ and for every process $p$, if (1) $p$ has passed its doorway at the end of $r$, (2) $r \leq r'$, (3) $p$ has not completed its operation during $(r' - r)$, and (4) at $(r' - r)$ there are (at least) $k$ steps which are strong (resp. weak) enabling steps, *then* $p$ is strongly (resp. weakly) enabled at the end of $r'$.

An algorithm that satisfies strong $k$-waiting, does not necessarily also satisfy weak $k$-waiting, and vice versa. To simplify the presentation, in the sequel, we will omit the type of a $k$-waiting progress condition (i.e., strong or weak), the type of an enabling step or the type of an enabling process, when it can be understood from the context or when the statement applies in both cases.

The $k$-waiting progress conditions capture the time a process may have to wait before it becomes enabled. Consider an implementation of a data structure which is protected by a single lock and assume that $n$ processes access the lock simultaneously. In such a scenario, each strong enabling step enables exactly one process to acquire the lock, complete its operation and release the lock. The last process captures the lock, after at least, strong $n-1$ enabling steps have been executed. Thus, such a lock-based data structure at best, satisfies strong $(n-1)$-waiting.

We point out that $k$-waiting does *not* guarantee that every process that has passed through its doorway becomes enabled no later than when $k$ other

processes have become enabled. The reason for that is that a single enabling step may cause several processes to become enabled. For a given $k$-waiting algorithm, the lower $k$ is, the higher is the potential that the algorithm, when executed, will exhibit a high concurrency behaviour. However, as in the case of using locks, algorithms which satisfy $k$-waiting for $k > 0$, may require processes to wait for one another, and thus, in some scenarios slow or stopped processes may prevent other processes from ever completing their operations.

In some scenarios wait-free algorithms (i.e., algorithms in which all the processes are always strongly enabled), preform better than lock-based algorithms and visa versa. For example, in scenarios when a process needs to hold a lock only for very short time, when there are no failures, no scheduling preemption and almost no page faults or cache misses, fine-grained lock-based algorithms might perform better. The decision whether to use a wait-free or a lock-based implementation depends on the assumption regarding the environment (i.e., the expectations regarding, failures, page faults, etc.). Similarly, one should not expect that for $k < k'$, a $k$-waiting algorithm would always (in all possible scenarios) preform better than the corresponding $k'$-waiting algorithm.

As in the case of wait-free or lock-based algorithms, when evaluating a $k$-waiting algorithm, it is not enough just to identify what progress condition it satisfies, it is also necessary to find out its time (step) complexity. The number of steps before and after enabling events can be arbitrary large, the $k$-waiting progress condition only gives an indication of how much time a process will have to wait without making progress and does not give any indication of its execution time while not waiting. Put another way, $k$-waiting is not intended to capture the overall time required for executing an operation, only the waiting time interval during the execution of an operation.

### 2.3 Simple examples

Below we give three examples of very simple algorithms, and for each algorithm we determine the smallest $k$ for which the algorithm is $k$-waiting. In the examples we use an atomic register called $x$. An atomic register supports atomic read and write operations. Thus, for example, the statement $x := x+1$ is not atomic and involves two operations: (1) reading the value of $x$, and then (2) writing a new value into $x$.

**Example 1.** An $(n-1)$-waiting algorithm: Code of process $i$ $(1 \leq i \leq n)$
**shared** $x$: atomic register, initially 1.

1 **wait until** $x = i$;
2 $x := x + 1$.

In Example 1, only process 1 is initially enabled, and process $i > 1$ becomes enabled only after process $i - 1$ executes line 2.

**Example 2.** A 0-waiting algorithm: Code of process $i$ $(1 \leq i \leq n)$
**shared** $x$: atomic register, initially 1.

1 **wait until** $x \geq 1$;
2 $x := x + 1$.

In Example 2, all the $n$ processes are initially enabled.

**Example 3.** A 1-waiting algorithm: Code of process $i$ $(1 \leq i \leq n)$
**shared** $x$: atomic register, initially 1.

1 **wait until** $(x = i$ or $x \geq 2)$;
2 $x := x + 1$.

In Example 3, only process 1 is initially enabled, and every process $i > 1$ becomes enabled after process 1 executes line 2.

## 3 Computational model and basic observations

Our model of computation consists of an asynchronous collection of $n$ deterministic processes that communicate via shared objects. Asynchrony means that there is no assumption on the relative speeds of the processes. In most of the cases we considered, the shared objects are registers which supports read and write operations. A register can be atomic or non-atomic. With an *atomic* register, it is assumed that operations on the register occur in some definite order. That is, reading or writing an atomic register is an indivisible action, and concurrent operations are ordered in their "real time" order. When reading or writing a non-atomic register, a process may be reading a register while another is writing into it, and in that event, the value returned to the reader is arbitrary. We will consider only atomic registers. In the sequel, by *registers* we mean *atomic* registers.

An event corresponds to an atomic step performed by a process. For example, the events which correspond to accessing registers are classified into two types: read events which may not change the state of the register, and write events which update the state of a register but do not return a value. A (global) state of an algorithm is completely described by the values of the registers and the values of the location counters of all the processes. A run is a sequence of alternating states and events.

A process executes correctly its algorithm until it (possibly) crashes. After it has crashed it executes no more steps. Given a run, a process that crashes is said to be *faulty* in that run, otherwise it is *correct*. In an asynchronous system there is no way to distinguish between a faulty and a very slow process. We will consider both the case where processes never fail and the case where processes may fail by crashing.

Several progress conditions have been proposed for data structures which avoid locking, and in which processes may fail by crashing. *Wait-freedom* guarantees that *every* active process will always be able to complete its pending operations in a finite number of steps [12]. *Non-blocking* (which is sometimes called lock-freedom) guarantees that *some* active process will always be able to

complete its pending operations in a finite number of steps [15]. *Obstruction-freedom* guarantees that an active process will be able to complete its pending operations in a finite number of steps, if all the other processes "hold still" long enough [13].

**Observation 1**

1. *An algorithm satisfies strong 0-waiting if and only if it satisfies wait-freedom.*

2. *An algorithm satisfies weak 0-waiting if and only if it satisfies obstruction-freedom.*

*Proof* (1) In a wait-free algorithm, by definition, every beginning process is strongly enabled. Thus, a wait-free algorithm satisfies strong 0-waiting. In a strong 0-waiting algorithm, by definition, every process that has passed its doorway is strongly enabled. Since the doorway is wait free, it follows that also every beginning process is strongly enabled. Thus, a strong 0-waiting algorithm satisfies wait-freedom. (2) In an obstruction-free algorithm, by definition, every beginning process is weakly enabled. Thus, an obstruction-free algorithm satisfies weak 0-waiting. In a weak 0-waiting algorithm, by definition, every process that has passed its doorway is weakly enabled. Since the doorway is wait free, it follows that also every beginning process is weakly enabled. Thus, a weak 0-waiting algorithm satisfies obstruction-freedom. □

Several progress conditions have been proposed for data structures, which may involve waiting in the context where processes never fail. *Livelock-freedom* guarantees that, in the absence of process failures, if a process is active, then *some* process must eventually complete its operation. A stronger property is *starvation-freedom* which guarantees that, in the absence of process failures, *every* active process must eventually complete its operation.

In a model where participation is required, every process must eventually become active and execute its code. A more interesting and practical situation is one in which participation is *not* required, as is usually assumed when solving resource allocation problems or when designing concurrent data structures. We *always* assume that participation is *not* required.

In general, wait-freedom is a strictly stronger progress condition than 0-waiting and starvation-freedom combined. However, this is not the case for $n = 2$.

**Observation 2** *Any weak 0-waiting starvation-free algorithm for two processes is wait-free.*

*Proof* Assume to the contrary that, there is a 0-waiting starvation-free algorithm for two processes which is not wait-free. Let the names of the two processes be $p$ and $q$. Since the algorithm is 0-waiting and starvation-free, it follows that,

– By 0-waiting, if only $p$ (resp. $q$) participates and $p$ (resp. $q$) does not fail then $p$ (resp. $q$) will eventually properly terminate.

- By 0-waiting, if both $p$ and $q$ participate and $p$ does not fail but $q$ fails then from some point on $p$ will run alone and will eventually properly terminate.
- By 0-waiting, if both $p$ and $q$ participate and $q$ does not fail but $p$ fails then from some point on $q$ will run alone and will eventually properly terminate.
- By starvation-freedom, if both $p$ and $q$ participate and none of them fails then both eventually properly terminate.

The fact that in all the above runs, a correct participating process always properly terminates, implies that the implementation also satisfies wait-freedom for two processes. That is, the above runs are exactly the runs in which correct processes are required to terminate when wait-freedom is assumed. A contradiction. □

## 4 Consensus and election

The *consensus* problem is to find a solution for $n$ processes, where each process starts with an input value from some domain, and must choose some participating process' input as its output. All $n$ processes together must choose the same output value. In the *election* problem one or more processes independently initiate their participation in an election to decide on a leader. Each participating process should eventually output either 0 or 1 and terminate. At most one process may output 1, and in the absence of faults exactly one of the *participating* processes should output 1. The process which outputs 1 is the elected leader.

### 4.1 A 2-waiting consensus algorithm

In [21], an election algorithm is presented, using $\lceil \log n \rceil + 1$ registers, which is correct under the following assumptions: (1) processes never fail, and (2) only the elected leader is required to terminate. A modified version of the election algorithm from [21], is used below for proving the following theorem,

**Theorem 1** *There are strong and weak 2-waiting starvation-free consensus algorithms and strong and weak 2-waiting starvation-free election algorithms for $n \geq 2$ processes, using $\lceil \log n \rceil + 2$ registers.*

The consensus algorithm presented below uses the shared registers *turn* and *decision* and the array of registers $V[1..\lceil \log n \rceil]$. All these registers are initially 0, except for the *decision* register which is initially $\perp$. Also, for each process, the local variables *level* and $j$ are used. The processes have unique identifiers. We will use the statement **await** *condition* as an abbreviation for **while** $\neg$*condition* **do** *skip*. The code of the algorithm appears in Figure 1.

The process that is last to write to *turn* (line 1) attempts to become the leader and to force all the other processes to decide on its input value. It does so, by waiting for each of the registers $V[j]$ to be 0 (lines 3-8) and then sets the register to its id (line 9). A process becomes the leader if it manages to

CODE OF PROCESS $p$'S PROGRAM WITH INPUT VALUE $input_p$

```
function consensus;
1   turn := p;
2   for level := 1 to ⌈log n⌉ do
3       repeat
4           if decision ≠ ⊥ then return(decision) fi;
5           if turn ≠ p then
6               for j := 1 to level − 1 do if V[j] = p then V[j] := 0 fi od;
7               await(decision ≠ ⊥); return(decision) fi
8       until V[level] = 0;
9       V[level] := p;
10      if turn ≠ p then
11          for j := 1 to level do if V[j] = p then V[j] := 0 fi od;
12          await(decision ≠ ⊥); return(decision) fi
13  od;
14  decision := input_p; return(decision)
end_function
```

**Fig. 1** A 2-waiting starvation-free consensus

write its id into all the registers during the period that *turn* equals its id. Any process that notices that *turn* is no longer equals its id, gives up on becoming the leader, and erase any write it has made (lines 6 & 11). The leader writes its input value into *decision*, and all the processes decide on that value.

There are runs of the algorithm in which every process manages to set $\lceil \log n \rceil$ registers before discovering that another process has modified *turn*, and as a result has to set back to 0 some of the registers before terminating. Proving the correctness of the algorithm is rather challenging, due to the existence of such runs.

It is straightforward to use the above consensus algorithm for solving election. Each process uses its identifer as its input. The value that all the processes decide on in the consensus algorithm, identifies the leader.

### 4.2 Proof of Theorem 1

The proof of Theorem 1 follows from the detailed correctness proof of the 2-waiting consensus algorithm given below. The proof of the consensus algorithm is an adaptation of the proof for the election algorithm from [21]. Part of the proof is similar to a proof of a similar algorithm that appeared in [26] in the context of fault tolerant algorithms.

The fact that the algorithm uses $\lceil \log n \rceil + 2$ registers is obvious from inspecting the algorithm. In the following, the *leader* is the process that writes its input value in to the *decision* register (line 14). A process is at level $k$, when the value of its private *level* register is $k$.

**Lemma 1 (liveness)** *In the absence of faults, at least one leader is elected.*

*Proof* Assume to the contrary that no leader is elected. Let $r$ be an infinite run with no faults where no leader is elected, and let $p$ be the last processes to write to *turn* in run $r$. Let $q$ be the process with the highest value of *level* when $p$ writes to *turn*. At some point $q$ will notice that $turn \neq q$, and set back to 0, all the entries of the array $V$ which equal to $q$. Repeat this argument with the new highest process. Thus, any entry of the array $V$ which process $p$ may wait on, will eventually be set back to 0, enabling $p$ to proceed until it is elected. A contradiction. □

We say that a process is at level $k$, when the value of its private *level* register is $k$. We say that a group of processes $P$ have noticed *together* that $V[k] = 0$, if each process in $P$: (1) is at level $k$, and (2) has notice that $V[k] = 0$ (when executing the until statement in line 8), before any other process in $P$ has written $V[k]$ (by executing the assignment in line 9).

**Lemma 2** *For any $k \in \{1, ..., \lceil \log n \rceil\}$ and for any group of processes $P$, if the processes $P$ have noticed together that $V[k] = 0$ then at most one process in $P$ can either (1) continue to level $k + 1$ or (2) change any register other than $V[k]$.*

*Proof* Assume that a set of processes, denoted $P$, are at level $k$, and they have noticed together that $V[k] = 0$. One of these processes, say $p \in P$, must be the last to update *turn*. If $k = 1$, each process in $P - \{p\}$ will notice that *turn* is different from its id (line 10), possibly write 0 into $V[1]$ (line 11), and wait & return (line 12).

Assume $k > 1$. To reach level $k$, each process in $P$ must have seen in all the levels smaller then $k$ that *turn* is equal to its id. Thus, *before* $p$ has set *turn* to its id, each of the other processes in $P$, must have seen in all the levels smaller then $k$ that *turn* is equal to its id.

Since the processes in $P$ have noticed together that $V[k] = 0$, by definition, it must be the case that before any of the processes in $P - \{p\}$ could execute the assignment at line 9, $p$ has already set $V[1], ..., V[k - 1]$ to its id. This implies that by the time each process in $P - \{p\}$ executes the statement in line 9, the following two conditions hold: (1) *turn* is different from its id, and (2) the values of the registers $V[1], ..., V[k - 1]$ are all different from its id. Thus, by the time each process in $P - \{p\}$ executes the if statement in line 10 it finds out that *turn* is different from its id, possibly writes 0 into $V[k]$ (line 11), and waits & returns (line 12), without a need to write 0 to any of the registers $V[1], ..., V[k - 1]$. Process $p$, may continue to level $k + 1$ or itself notices that $turn \neq p$ and sets some or all of the registers $V[1], ..., V[k - 1]$ to 0, but it is the only process, among the processes in $P$, that may set any shared register other than $V[k]$. □

**Lemma 3 (safety)** *At most one leader is elected.*

*Proof* For proving the lemma, an accounting system of credits is used. Initially, the number of credits is $2n - 1$. New credits can not be created during the execution of the algorithm. The credit system ensures that a process acquires

exactly $2^{k-1}$ credits before it can reach level $k$. Being elected is equivalent to reaching level $\log n + 1$. Thus, the credit system ensures that a process must acquire $2^{\log n+1-1} = n$ credits before it can be elected. Once a process is elected, it may not release any of its credits. Thus, it is not possible for two processes to get elected.

Without loss of generality it is assumed that $n$, the number of processes, is a power of 2. Initially, each process holds 1 credit, and each register $V[k]$ where $1 \leq k \leq \log n$ holds $2^{k-1}$ credits. Thus, the total number of credits is $n + \sum_{k=1}^{\log n} 2^{k-1} = 2n - 1$. As a results of an operation taken by a process credits may be transferred from a register to a process and vice versa, and between processes. We list below 4 rules which capture all possible operations by processes and their effect:

1. **No transfer of credits:** No credits are transferred when a process (1) checks the value of a register, (2) writes into *turn*, or (3) executes a *return* statement.
2. **Transferring credits between a register and a process:** When a process writes its id into register $V[k]$, changing $V[k]$'s value from 0 to its id, $2^{k-1}$ credits are transferred from $V[k]$ to that process. When a process writes 0 into register $V[k]$ which does not already holding 0, $2^{k-1}$ credits are transferred to $V[k]$ from that process.
   *Remark:* This is the only rule for transferring credits between a register and a process. Initially, $V[k]$ holds $2^{k-1}$ credits, so the first time $V[k]$'s value changes, it has enough credits to transfer. Before any subsequent transfer from $V[k]$ to a process, its value has to be set back to 0, and each time this happens $V[k]$ gets back $2^{k-1}$ credits. So, $V[k]$ always has enough credits to transfer to a process that changes $V[k]$'s value from 0 to its id (line 9). A process at level $k$ may changes $V[k]$'s value back to 0 at most once (line 11). Under the assumption, which we justify later, that the credit system ensures that a process acquires exactly $2^{k-1}$ credits before it can reach level $k$, and that these $2^{k-1}$ credits are not used for something else, a process always has enough credits to transfer to $V[k]$ if it changes $V[k]$'s value to 0.
3. **Transferring credits between processes when moving to an upper level:** Let $P$ be a maximal[2] set of processes that have noticed together that $V[k] = 0$. By Lemma 2, at most one process from $P$ can continue to level $k + 1$. Assume process $p \in P$ continues to level $k + 1$. We consider two cases:
   - At level $k$, when executing line 9, process $p$ changes $V[k]$'s value from 0 to its id. By rule 2, $2^{k-1}$ credits are transferred from $V[k]$ to $p$. Thus, $p$ has $2^k$ credits available, $2^{k-1}$ credits from reaching level $k$, plus $2^{k-1}$ credits from $V[k]$, giving $p$ the total of $2^k$ credits it needs for level $k+1$.
   - At level $k$, when executing line 9, process $p$ does not change $V[k]$'s value from 0 to its id. This implies that there must be another process $q \in P$

[2] A set of processes $P$ is maximal with respect to property $\phi$, if (1) $P$ satisfies $\phi$, and (2) there is no set $Q$, such that $P \subset Q$ and $Q$ satisfies $\phi$.

that, before $p$ has executed line 9, was the last process to change $V[k]$'s value back to 0. By rule 2, $2^{k-1}$ credits are transferred from $V[k]$ to $q$. Thus, $q$ has $2^k$ credits available, $2^{k-1}$ credits from reaching level $k$, plus $2^{k-1}$ credits from $V[k]$. In this case, immediately after $p$ executes line 9, $2^{k-1}$ credits are transferred from $q$ to $p$, leaving $q$ with $2^{k-1}$ credits and giving $p$ the total of $2^k$ credits ($2^{k-1}$ credits from reaching level $k$, plus $2^{k-1}$ credits from $q$) it needs for level $k+1$.

By Lemma 2, each process in $P$ (including $q$) that does not continue to the level $k+1$ can only execute $V[k] := 0$ (line 11), transferring (by Rule 2) to $V[k]$ the $2^{k-1}$ credits it has by getting this far, if it succeeds in changing $V[k]$'s value back to 0.

4. **Transferring credits between processes without moving to an upper level:** Let $P$ be a maximal set of processes that have noticed together that $V[k] = 0$, and assume that no process in $P$ continues to level $k+1$. By Lemma 2, at most one process in $P$, say process $p$, can change any register other than $V[k]$. We consider two cases:
   - At level $k$, when executing line 9, process $p$ changes $V[k]$'s value from 0 to its id. By rule 2, $2^{k-1}$ credits are transferred from $V[k]$ to $p$, Thus, $p$ has $2^k$ credits available, $2^{k-1}$ credits from reaching level $k$, plus $2^{k-1}$ credits from $V[k]$.
   - At level $k$, when executing line 9, process $p$ does not change $V[k]$'s value from 0 to its id. This implies that there must be another process $q \in P$ that, before $p$ has executed line 9, was the last process to change $V[k]$'s value back to 0. By rule 2, $2^{k-1}$ credits are transferred from $V[k]$ to $q$, giving $q$ a total of $2^k$ credits. In this case, immediately after $p$ executes line 9, $2^{k-1}$ credits are transferred from $q$ to $p$, leaving $q$ with $2^{k-1}$ credits, and giving $p$ a total of $2^k$ credits ($2^{k-1}$ credits from reaching level $k$, plus $2^{k-1}$ credits from $q$).

   Setting to 0 every variable from $V[1]$ to $V[k]$ accounts for $2^k - 1$ credits (i.e., $\sum_{i=1}^{k} 2^{i-1} = 2^k - 1$), so (in both cases) $p$ has enough credits and no new credits should be created by $p$ when it sets to 0 multiple registers.

   By Lemma 2, each process in $P - \{p\}$ (including $q$) can only execute $V[k] := 0$ (line 11), transferring (by Rule 2) to $V[k]$ the $2^{k-1}$ credits it has by getting this far, if it succeeds in changing $V[k]$'s value back to 0.

Given the above description of the accounting system, we can now justify the following two claims made earlier:

1. *A process acquires exactly $2^{k-1}$ credits before it can reach level $k$.*
   This is proven by induction on the level $k$. For $k = 1$, the claim follows immediately from the fact that initially each process has one credit. We assume that the claim holds for level $k$ and prove that it also holds for level $k+1$. By Rule 3, before process $p$ moves to level $k+1$, it gets additional $2^{k-1}$ credits either from $V[k]$ or from another process. Thus, $p$ has $2^k$ credits available, $2^{k-1}$ credits by the induction hypothesis (from reaching level $k$) plus $2^{k-1}$ as explained above, giving $p$ the total of $2^k$ credits it needs for level $k+1$.

2. *No new credits are created.*
   As already explained in Rule 2, $V[k]$ always has enough credits to transfer to a process that changes $V[k]$'s value from 0 to its id. Furthermore, since the credit system ensures that a process acquires exactly $2^{k-1}$ credits before it can reach level $k$, and since these $2^{k-1}$ credits are not used for something else, a process at level $k$ always has enough credits to transfer to $V[k]$ if it changes $V[k]$'s value to 0.
   By Lemma 2, at most one process in a maximal set of processes $P$ that have noticed together that $V[k] = 0$, say process $p$, can change any register other than $V[k]$. By Rule 4, $p$ has $2^k$ credits available, $2^{k-1}$ credits from reaching level $k$ plus $2^{k-1}$ from either $V[k]$ or from another process. As already explained in Rule 4, setting to 0 every variable from $V[1]$ to $V[k]$ accounts for $2^k - 1$ credits (i.e., $\sum_{i=1}^{k} 2^{i-1} = 2^k - 1$), so (in both cases) $p$ has enough credits and no new credits should be created by $p$ when it sets to 0 multiple registers.

As already mentioned, initially, the number of credits is $2n-1$. No new credits are created, and a process must acquire $n$ credits before it can be elected. Once a process is elected, it may not release any of its credits. Thus, it is not possible for two processes to get elected. □

**Theorem 2 (agreement & validity)** *All the participating processes decide on the same value, and this decision value is the input of a participating process.*

*Proof* It follows from Lemma 1 and Lemma 3, that exactly one leader is elected. The leader will eventually write its input value into the *decision* register, and all the participating will decide on that value. □

**Theorem 3 (starvation-freedom)** *In the absence of faults, every participating process eventually terminates.*

*Proof* Once a leader is elected and sets the *decision* register to its input value, all correct participating processes will eventually find out that $decision \neq \perp$ and properly terminate. In the absence of faults, by Lemma 1, at least one leader is eventually elected, and thus all the participating processes eventually terminate. □

**Theorem 4 (2-waiting)** *The consensus algorithm satisfies strong and weak 2-waiting.*

*Proof* In every run there are at most two (strong or weak) enabling events. The first is the event after which the leader becomes enabled. (This can happen at most once since being enabled is a stable property.) The second event is when the leader sets the *decision* register to its input value (line 14), after which all the other processes immediately become enabled. In fact, in this particular algorithm in every run in which some process terminates, there are exactly two enabling events. Consider for example a run where process $p$ runs alone until

it is elected and terminates. The first enabling event is when the local variable *level* of $p$ equals $\lceil \log n \rceil$ and $p$ reads in Line 10 that $turn = p$. Before that read event all the processes are disabled, and after that read event $p$ becomes (strongly) enabled (and all the other processes are still disabled). Once process $p$ executes line 14, all the processes become enabled even though they haven't started yet. Hence, the consensus algorithm satisfies 2-waiting.

We point out that a process does not necessarily become weakly enabled after taking its first step. To see that, recall that being weakly enabled is a stable property. Assume that process $p$ wakes up, runs alone, is elected but is suspended before setting *decision* to its input (in Line 14). At that point all the other processes, regardless of the number of steps they have taken so far, are disabled. Once process $p$ executes line 14 all the other processes become enabled (also those that haven't taken any steps yet). □

This completes the proof of Theorem 1.

### 4.3 An impossibility result

In [21], it has been proven that, even in the absence of faults, any election algorithm for $n$ processes must use at least $\lceil \log n \rceil + 1$ registers. This lower bound holds also for consensus. Thus, Theorem 1 provides an almost tight space upper bound. It is known that there are no wait-free consensus or election algorithms, using registers [9,12,18,19]. Below we slightly generalize these known impossibility results for wait-free consensus and election.

**Theorem 5** *There are no weak 0-waiting starvation-free consensus or election algorithms for $n \geq 2$ processes, using registers.*

*Proof* The result follows from Observation 2 and the known impossibility results that there are no *wait-free* consensus and election algorithms for two (or more) processes, using registers [9,12,18,19]. □

## 5 Adaptive renaming

### 5.1 The problem

The renaming problem allows processes, with distinct initial names from a large name space, to get distinct new names from a small output name space. In the non-adaptive version of the problem, the size of the new name space is a function of $n$, the total number of processes. Adaptive renaming is more demanding: the size of the new name space must be a function of the actual number of the participating processes.

An *adaptive $f(m)$-renaming* algorithm allows $m$ participating processes with initially distinct names from a large name space to acquire distinct new names from the set $\{1, ..., f(m)\}$. A *one-shot* renaming algorithm allows each

process to acquire a distinct new name just once. A *long-lived* renaming algorithm allows processes to repeatedly acquire distinct names. We focus below on solving one-shot adaptive renaming.

### 5.2 The algorithm

It is known that there is a wait-free adaptive $(2m-1)$-renaming algorithm using registers, where $m$ is the number of participating processes [6]. Below we extend this result to cover cases where waiting is possible.

**Theorem 6** *For any $1 \leq k < n$, there is a strong $(k+1)$-waiting starvation-free adaptive $(\max\{m, 2m-k-1\})$-renaming algorithm, where $1 \leq m \leq n$ is the number of participating processes, using registers.*

*Proof* For $1 \leq i \leq k$, let $E_i$ be the implementation of a strong 2-waiting election object from registers, from the proof of Theorem 1. Each process, say $p$, scans the $k$ election objects, $E_1, ..., E_k$, in order, starting with $E_1$. At each step, process $p$ tries to get elected, and either moves to the next election object if the returned value is 0 (i.e, not elected), or stops when the returned value is 1 (i.e, elected). If process $p$ stops on one of the $k$ election objects, then it is assigned the name that equals to the index of the election object on which it is elected. (I.e., if it stopped on $E_i$ then it is assigned name is $i$.). Otherwise, if all its operations on the election objects have returned 1 (which means that $k$ other processes already got the names 1 through $k$), process $p$ participates in a wait-free adaptive $(2m-1)$-renaming algorithm which uses registers only. Let $v$ be the value assigned to $p$ by the optimal renaming algorithm, then process $p$ is assigned the final new name $k+v$. Clearly, only $m-k$ processes will participate in the adaptive wait-free renaming, and thus $v \in \{1, ..., 2(m-k)-1\}$. This proves that the name name space is as stated in the Lemma. Next we prove that the algorithm satisfies $k+1$-waiting. There are two possible cases:

1. A process, say $p$, acquires a new name $i \leq k$. This means that $p$ got elected in $E_k$. So, at some point there was a strong enabling step which made $p$ strongly enabled after which it got elected at $E_k$. *Before* that strong enabling step, there where at most $k-1$ other strong enabling steps which strongly enabled $k-1$ other processes to get elected in objects $E_1, ..., E_{k-1}$, a total of $k$ strongly enabling events.
2. A process, say $p$, acquires a new name $i > k$. This means that $p$ acquired a name while participating in a wait-free adaptive $(2m-1)$-renaming algorithm. So, at some point there was a strong enabling step which made $p$ strongly enabled after which it acquired a name. *Before* that strongly enabling step, there where at most $k$ other strong enabling steps which strongly enabled $k$ other processes to get elected in objects $E_1, ..., E_k$, a total of $k+1$ strongly enabling events.

We notice that after the $k$'th enabling step, the step which made the process that got elected in $E_k$ enabled, the next enabling step simultaneously made *all* the remaining processes enabled. □

The result stated in Theorem 6 holds, with almost the same proof, if we replace the word *strong* with *weak* in the statement of the theorem.

### 5.3 A name-space lower bound

A wait-free adaptive $(2m-1)$-renaming algorithm using registers, where $m$ is the number of participating processes is called an *optimal* adaptive renaming algorithm w.r.t. registers, because it matches the known lower bound on the name space. This known lower bound can be easily derived from the known impossibility result for set-consensus [4,14,20]. Below we slightly generalize this lower bound result.

**Theorem 7** *There is no weak* 0*-waiting starvation-free adaptive* $\max\{1, 2m-2\}$*-renaming algorithm, where* $m$ *is the number of participating processes, using registers.*

*Proof* The result follows from Observation 2 and the known impossibility result that there is no *wait-free* adaptive $m$-renaming algorithm for two processes, using registers [4,14,20]. □

## 6 Mutual exclusion

The mutual exclusion problem is to design an algorithm (i.e., a lock) that guarantees mutually exclusive access to a critical section among $n$ competing processes [7]. It is assumed that each process is executing a sequence of instructions in an infinite loop. The instructions are divided into four continuous sections: the remainder, entry, critical and exit. The entry section consists of two parts: the *doorway* which is *wait-free*, and the waiting part which includes one or more loops. A *waiting* process is a process that has finished its doorway code and reached the waiting part, and a *beginning* process is a process that is about to start executing its entry section. Like in the case of the doorway, the exit section is also required to be wait-free. It is assumed that processes do not fail, and that a process always leaves its critical section.

### 6.1 Definitions

The *mutual exclusion problem* is to write the code for the entry and the exit sections in such a way that the following *two* basic requirements are satisfied.

***Livelock-freedom:*** *If a process is trying to enter its critical section, then some process, not necessarily the same one, eventually enters its critical section.*

***Mutual exclusion:*** *No two processes are in their critical sections at the same time.*

Satisfaction of the above two properties is the minimum required for a mutual exclusion algorithm. For an algorithm to be fair, satisfaction of an additional condition is required.

***First-in-first-out (FIFO):*** *A beginning process cannot execute its critical section before a waiting process completes executing its critical section.*

*Linear-waiting* is a requirement which is slightly weaker than FIFO. Linear-waiting means that no (beginning or not) process can execute its critical section twice while some other process is kept waiting.

### 6.2 Three observations

We prove below three observation regarding the level of waiting needed when solving the mutual exclusion problem.

**Theorem 8** *(1) There is no strong* $(n-2)$*-waiting livelock-free mutual exclusion algorithm; (2) There are strong* $(n-1)$*-waiting FIFO mutual exclusion algorithms using strong synchronization primitives; (3) There are strong* $n$*-waiting FIFO mutual exclusion algorithms using registers.*

*Proof* (1) Let $A$ be an arbitrary mutual exclusion algorithm. Assume that $n$ processes are trying to enter their critical sections of $A$ simultaneously, and they have all passed their doorways. In such a scenario, each strong enabling step enables exactly one process to enter its critical section, complete its operation and release the lock. The last process enters its critical section, after at least strong $n-1$ enabling steps have been executed. Thus, at best, $A$ satisfies strong $(n-1)$-waiting, but it does not satisfy $(n-2)$-waiting. (2) Anderson's queue-based algorithm [1], which uses registers and fetch-and-increment object, is an example of a strong $(n-1)$-waiting FIFO mutual exclusion algorithm. (3) The FIFO mutual exclusion algorithm from [17] use only registers and satisfies strong $n$-waiting. □

In the context of mutual exclusion, a process is weakly enabled if and only if it is strongly enabled. Thus, the result stated in Theorem 8 holds, if we replace the word *strong* with *weak* in the statement of the theorem.

## 7 Fairness

Fairness requirements guarantee that a process will not bypass another process "too many times". The problem of implementing a *k-fair* data structure is to write the code of each operation in such a way that the following requirement is satisfied,

> **$k$-fairness:** No beginning process can complete $k+1$ operations while some other process which has already passed the doorway of some operation has not completed the operation yet.

The term first-in-first-out (FIFO) is used for 0-fairness. For every $k \geq 1$, $k$-fairness *does not* imply livelock-freedom. We address the following question: When is it possible to transform a non-blocking data structure into the corresponding fair data structure? We show that, when only registers are used, such a transformation *must involve waiting.*

**Theorem 9** *For any $k \geq 0$, it is not possible to automatically transform every data structure, which has a non-blocking implementation using registers, into the corresponding $k$-fair non-blocking data structure, using registers.*

*Proof* For any $k \geq 0$, a data structure that satisfies both $k$-fairness and non-blocking must also satisfy wait-freedom. In [10], it is shown that there exists an object which has a non-blocking implementation using registers, but does not have a wait-free implementation using registers. The existence of such an object implies that it is not possible to automatically transform every non-blocking data structure into the corresponding wait-free data structure using only registers. The result follows. □

**Theorem 10** *It is possible to automatically transform every non-blocking data structure, using only registers, into the corresponding strong 1-waiting data structure which (1) satisfies 1-fairness and starvation-freedom, and (2) guarantees that the execution of the doorway of each operation requires a constant number of steps.*

*Proof* It was recently proved in [24] that, using registers, it is possible to automatically transform any non-blocking data structure into the corresponding starvation-free data structure which satisfies the following three properties: (1) no beginning process may complete two operations before another process that has passed its doorway completes its operation; (2) All the processes that have passed their doorways and are not strongly enabled, eventually become strong enabled at the same time; (3) the execution of the doorway requires only three steps, in which only registers are accessed. This transformation is called the *fair synchronization algorithm.* Property (1) above means that the transformed data structure satisfies 1-fairness; property (2) implies that it satisfies strong 1-waiting. The result follows. □

## 8 Related work

In [8], it is suggested to model contention at a shared object with the help of stall operations. In the case of simultaneous accesses to a single memory location, only one operation succeeds, and other pending operations must stall. The measure of contention is the worst-case number of stalls that can be induced by an adversary scheduler. Our study of the new progress conditions complements the study of the complexity measure of [8].

As already mentioned, the following important progress conditions have been proposed for data structures which avoid waiting: wait-freedom [12], non-blocking [15], and obstruction-freedom [13]. Symmetric and asymmetric

progress conditions are studied in [16,23]. In [11], the authors identify an interesting relationship that unifies six progress conditions ranging from the deadlock-free and starvation-free conditions common to lock-based systems, to the obstruction-free, non-blocking and wait-free conditions common to lock-free systems.

The impossibility result that there is no consensus algorithm that can tolerate even a single crash failure was first proved for the asynchronous message-passing model in [9], and later has been extended for the shared memory model with atomic registers, in [18]. A comprehensive discussion of wait-free synchronization is given in [12].

In [21] it is proved that, in the absence of failures, $\lceil \log n \rceil + 1$ registers are necessary and sufficient for election, assuming that only the elected leader is required to ever terminate. We use the key ideas from [21], in our implementations of the 2-waiting starvation-free consensus and election algorithms. The one-shot renaming problem was first solved for message-passing systems [2], and later for shared memory systems [3]. In [5] a long-lived wait-free renaming algorithm was presented. Many of the results on renaming are discussed in [6].

The mutual exclusion problem was first stated and solved for $n$ processes by Dijkstra in [7]. Numerous solutions for the problem have been proposed since it was first introduced in 1965 [22]. In [24], it is shown that it is possible to automatically transfer any non-blocking or wait-free data structure into a similar data structure which satisfies a strong fairness requirement, without using locks and with limited waiting.

## 9 Discussion and open problems

We have introduced a new set of progress conditions, called $k$-waiting, for $k \geq 0$. The new conditions are intended to quantitatively capture the "amount of waiting" of processes in asynchronous concurrent algorithms. To illustrate the utility of the new conditions, we have derived lower and upper bounds, and impossibility results for well-known basic problems such as consensus, election, renaming and mutual exclusion. We also presented some results regarding the relation between waiting and fairness. Much, however, remains to be done.

The new progress conditions together with our technical results, indicate that there is an interesting area of concurrent algorithms that deserve further investigation. A few specific interesting open problems are: Are there 1-waiting starvation-free consensus and election algorithms for $n \geq 2$ processes, using registers? Is the upper bound of Theorem 6, on the name space for $k$-waiting starvation-free adaptive renaming, tight? It would also be interesting to look at various variants of $k$-waiting.

To conclude, we have focused on identifying some intermediate notion of waiting, and the basic definition of $k$-waiting appears to make sense as a candidate definition. The various results presented, provide some evidence that this is a good definition. We hope that our conceptual contributions will lead to interesting conversations and further results regarding this unexplored area.

## References


1. T. E. Anderson. The performance of spin lock alternatives for shared-memory multiprocessor. *IEEE Trans. on Parallel and Distributed Systems*, 1(1):6–16, January 1990.
2. H. Attiya, A. Bar-Noy, D. Dolev, D. Koller, D. Peleg, and R. Reischuk. Renaming in an asynchronous environment. *Journal of the Association for Computing Machinery*, 37(3):524–548, July 1990.
3. A. Bar-Noy and D. Dolev. Shared memory versus message-passing in an asynchronous distributed environment. In *Proc. 8th ACM Symp. on Principles of Distributed Computing*, pages 307–318, 1989.
4. E. Borowsky and E. Gafni. Generalized FLP impossibility result for $t$-resilient asynchronous computations. In *Proc. 25th ACM Symp. on Theory of Computing*, pages 91–100, 1993.
5. J.E. Burns and G.L. Peterson. The ambiguity of choosing. In *Proc. 8th ACM Symp. on Principles of Distributed Computing*, pages 145–158, August 1989.
6. A. Castaneda, S. Rajsbaum, and M. Raynal. The renaming problem in shared memory systems: An introduction. *Computer Science Review*, 5(3):229–251, 2011.
7. E. W. Dijkstra. Solution of a problem in concurrent programming control. *Communications of the ACM*, 8(9):569, 1965.
8. C. Dwork, M. P. Herlihy, and O. Waarts. Contention in shared memory algorithms. *Journal of the ACM*, 44(6):779–805, 1997.
9. 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, 1985.
10. M. Herlihy. Impossibility results for asynchronous pram. In *Proc. of the 3rd Annual ACM Symp. on Parallel Algorithms and Architectures*, pages 327–336, 1991.
11. M. Herlihy and N. Shavit. On the nature of progress. In *15th International Conference on Principles of Distributed Systems (OPODIS 2011)*, 2011. *LNCS 7109* Springer Verlag 2011, 313-328.
12. M. P. Herlihy. Wait-free synchronization. *ACM Trans. on Programming Languages and Systems*, 13(1):124–149, January 1991.
13. M. P. Herlihy, V. Luchangco, and M. Moir. Obstruction-free synchronization: Double-ended queues as an example. In *Proc. of the 23rd International Conference on Distributed Computing Systems*, page 522, 2003.
14. M. P. Herlihy and N. Shavit. The topological structure of asynchronous computability. *Journal of the ACM*, 46(6):858–923, July 1999.
15. M. P. Herlihy and J. M. Wing. Linearizability: a correctness condition for concurrent objects. *ACM Trans. on Programming Languages and Systems*, 12(3):463–492, 1990.
16. D. Imbs, M. Raynal, and G. Taubenfeld. On asymmetric progress conditions. In *Proc. 29th ACM Symp. on Principles of Distributed Computing*, pages 55–64, 2010.
17. L. Lamport. A new solution of Dijkstra's concurrent programming problem. *Communications of the ACM*, 17(8):453–455, August 1974.
18. M.C. Loui and H. Abu-Amara. Memory requirements for agreement among unreliable asynchronous processes. *Advances in Computing Research*, 4:163–183, 1987.
19. S. Moran and Y. Wolfstahl. Extended impossibility results for asynchronous complete networks. *Information Processing Letters*, 26(3):145–151, 1987.
20. M. Saks and F. Zaharoglou. Wait-free $k$-set agreement is impossible: The topology of public knowledge. *SIAM Journal on Computing*, 29, 2000.
21. E. Styer and G. L. Peterson. Tight bounds for shared memory symmetric mutual exclusion problems. In *Proc. 8th ACM Symp. on Principles of Distributed Computing*, pages 177–191, August 1989.
22. G. Taubenfeld. *Synchronization Algorithms and Concurrent Programming*. Pearson / Prentice-Hall, 2006. ISBN 0-131-97259-6, 423 pages.
23. G. Taubenfeld. The computational structure of progress conditions. In *24th international symposium on distributed computing (DISC 2010)*, Sept. 2010. *LNCS 6343* Springer Verlag 2010, 221–235.
24. G. Taubenfeld. Fair synchronization. In *Journal of Parallel and Distributed Computing* 97:1–10, November 2016. (Also in: *LNCS 8205*, 2013, 179–193, DISC 2013.)

25. G. Taubenfeld. Waiting in Concurrent Algorithms. In *4th international conference on networked systems (NETYS 2016)*, Marrakech, Morocco, May 2016. *LNCS 9944*, 2016, 345–360.
26. G. Taubenfeld. A closer look at fault tolerance. In *Theory of Computing Systems* (2017), To appear. (Also in: Proceedings of PODC 2012, 261–270.)