Journal of Universal Computer Science, vol. 2, no. 1 (1996), 2-19
submitted: 1/8/95, accepted: 23/1/96, appeared: 28/1/96

Wait-Freedom vs. Bounded Wait-Freedom in Public Data Structures

Hagit Brit
Computer Science Department, Technion, Haifa 32000, Israel.

Shlomo Moran
Computer Science Department, Technion, Haifa 32000, Israel.

Abstract: In this paper we define and study *public data structures*, which are concurrent data structures in the shared memory environment, which enable access to an unknown (and possibly infinite) set of identical processes. Specific cases of such data structures (like counting networks and concurrent counters) have been studied recently, and such data structures seem to model concurrent systems like client-server applications, in which the identities of the clients, and sometimes also their number, are not known apriori.
Specifically, we study the relation between wait-free and bounded wait-free public data structures - the former guarantees that every operation performed on the data structure always terminates, regardless of the relative speed of the processes; the latter guarantees that every such operation is terminated within a fixed number of steps. We present an example of a public data structure which is wait-free but not bounded wait-free, and then we show that if all the concurrent objects of the data structure are *periodic*, then wait-freedom implies bounded wait-freedom.

1 Introduction

The subject of concurrent data structures has been the focus of several recent works, which are motivated by the development of new parallel computers. A traditional implementation of a (sequential) data structure consists of the codes for all the operations the data structure supports, which behaves correctly when all the operations are executed one after the other in a sequential fashion. An implementation of a concurrent data structure gives a code which must behave correctly even when executed by many processes concurrently.

Of particular interest are *wait-free* concurrent data structures, which guarantee that any operation by a process is completed within a finite number of steps, regardless of the behavior of other processes (such as abnormal termination). In implementing concurrent structures, one usually assumes that the total number of processes in the system, as well as the identities of these processes, are known. However, this assumption is not always valid: for instance, in common client-server applications, the identities of the clients, and in some cases also their number, are not known a priori. Hence we define the notion of a *public data structure*. A public data structure is a concurrent data structure that is required to work correctly for any finite number of concurrent processes – nothing is assumed in advance about the number or the identity of the processes that might access it. Among the data structures studied in the literature, counting networks and concurrent counters [AHS91, MTY95] are public data structures.

In this work we distinguish between two wait-freedom requirements: A weaker one, which requires that every operation on the data structure always terminates within a finite number of steps, and a stronger one, which requires that every such operation always terminates within a fixed and pre-determined number of steps. A data structure which satisfies the latter property is called *bounded* wait-free, and a wait-free data structure which is not bounded wait-free is called *unbounded* wait-free. Bounded wait-freedom is superior to (possibly unbounded) wait-freedom for both practical and theoretical reasons. Counting networks [AHS91], and other types of concurrent counters [MTY95], are examples of bounded wait-free data structures. In [BMT95] we use the fact that wait-free concurrent counters must be bounded wait-free, to show that a concurrent counter which counts modulo m may be constructed from counters which count modulo $b_1 \cdots, b_n$ respectively, only if every prime factor of m is also a prime factor of one of the b_i's.

The relation between wait-freedom and bounded wait-freedom in concurrent data structures was first studied in [Her91a], where a hierarchy of wait-free classes is given. In particular, [Her91a] brings an example of an unbounded wait-free data structure, which is defined by the approximate agreement task. The example in [Her91a] is based on the fact that the corresponding data structure has infinitely many initial states. It should be noted that if a data structure (a) has finitely many initial states, (b) can be accessed by a bounded number of processes, and (c) each process may perform a bounded number of (non-atomic) operations, then being wait-free is equivalent to being bounded wait-free, as can be shown by the Infinity Lemma [Kön36]. A well known example of such case is the consensus problem [FLP85, CIL87]. We present in this paper a simple example that shows that this is not the case for public data structures. Then we show that if the shared objects used by a protocol satisfy certain conditions, wait-freedom becomes equivalent to bounded wait-freedom also in public data structures.

1.1 The computational model

Our model of computation consists of a collection of fully asynchronous identical deterministic processes that communicate via *atomic concurrent objects*. We model atomic concurrent objects by Mealy machines [HU79], where the input alphabet is the set of operations applicable to the object, and the output alphabet is the set of output values returned by the object. The objects are atomic in the sense that in every execution all the accesses to a given object are totally ordered in time.

1.2 Related Work

The area of concurrent and distributed data structure is relatively new, but has already drawn the attention of many researchers. While the term *concurrent* data structure, refers to a data structure that is stored in shared memory, the term *distributed* data structure refers to a collection of local data structures stored at different processors in a message passing system. We will not try to review all the relevant work here, but rather give just few pointers to the literature.

Few works have introduced general methods for transforming a given sequential implementation (one that works for just one process) into a wait-free concurrent one [Her91b, Plo89]. These results are mainly of theoretical interest since the constructions involved are too inefficient to be practical. Other transformations are introduced in [Her90] for a large class of structures using the compare-and-swap synchronization primitive; in [Her91b] using the load_link and store_conditional primitives; and in [AT93] using timing assumptions.

More efficient (though not necessarily wait-free) constructions for specific data structures have been proposed. Peterson [Pet83] and Lamport in [Lam86b, Lam86a] study wait-free implementation of reading and writing to atomic registers. Many constructions of concurrent B-trees, have been implemented mainly for use in databases, see for example [BS77, LY81, Sag85]. AVL trees, 2-3 trees, and a distributed extendible hash file have been implemented in [Ell80a, Ell80b, Ell85]. A distributed dictionary structure is studied in [Pel90]. A wait-free implementation of a queue where one enqueuing operation can be executed concurrently with one dequeuing operation is given in [Lam83]. An implementation of a queue that allows an arbitrary number of concurrent queuing and dequeuing operations is given in [HW87], the implementation is deadlock-free but allows starvation of individual processes. A wait-free implementation of union-find structures is described in [AH91]. An efficient wait-free implementation of a priority queue is given in [IR93]. These data structures are not public data structures, as they all assume a fixed and known set of processes which may access the data structures.

Aspnes, Herlihy and Shavit [AHS91] have implemented a counter which is a public data structure. They named the implementations they have found *counting networks*. Counting networks have been further investigated in [AA92, AHS91, HLS92, HSW91, KP92]. Another related public data structure, called *concurrent counter*, was introduced in [MTY95] and later studied in [MT93, BMT95].

Most of the works on wait-free protocols do not explicitly distinguish between wait-freedom and bounded wait-freedom. For example the definition of wait-freedom in [AHS91, IR93, MTY95] corresponds to our definition of bounded wait-freedom. The definition of wait-freedom in [Her91b] corresponds to our definition of wait-freedom, and is distinguished from bounded wait-freedom. Specifically, [Her91b] presents bounded wait-free implementations of concurrent object X using concurrent object Y, and proves impossibility results using the (weaker) wait-freedom requirement.

1.3 Preview

In the next section we provide the definitions used in the paper. Then we present a very simple public data structure, which uses two atomic bits, which is wait-free but not bounded wait-free. The following sections are devoted to presenting the main result of the paper, which present conditions, in terms of properties of the atomic objects used, which guarantee equivalence of wait-freedom and bounded wait-freedom for public data structures. First we define *periodic* data structures, and prove that periodic data structures satisfy an interesting property, called the *reconstruction* property. We then use this last result to show that every periodic wait-free data structure which has finite memory, must be bounded wait-free. We conclude by presenting some extensions of our main result, and by showing that further generalization of it might not be easy.

2 Definition and Notations

2.1 Concurrent Data Structures

A *concurrent system* consists of a shared memory and a collection of processes. The shared memory is modelled by a collection of *atomic objects*, and the processes are infinite state machines, defined by the set of procedures they can perform.

An *atomic object* is a basic memory unit, which enables processes to perform predefined atomic operations on it. Formally, an atomic object r is defined by a triple (G_r, D_r, O_r) where:

D_r is a set of the states of the atomic object.
O_r is a set of output values which can be returned by the object.
G_r is a set of atomic operations defined on the atomic object; each operation *op* in G_r is defined by a pair (δ, λ), where δ is the *transition function* of *op* and λ is the *output function* of *op*, as follows:
  $\delta : D_r \rightarrow D_r$ defines the new state of the object, as a function of its current state.
  $\lambda : D_r \rightarrow O_r$ defines the value returned by the object, as a function of its current (i.e., old) state.

For example, a v-valued read-modify-write (*rmw*) register, which may hold values from a set V of v elements, is an atomic object r defined by $r = (D_{rmw}, O_{rmw}, G_{rmw})$, where $D_{rmw} = O_{rmw} = V$, and G_{rmw} consists of the v^v functions from V to itself. If r is a read/write register over the same set of values V, then it has the same sets of states and output values as above, but its operation set consists of only $v + 1$ operations: a *read* operation and v *write* operations (one write operation for each value in V).

A *Concurrent data structure* over a concurrent system is a data structure that enables several processes to perform simultaneously non-atomic operation on it. Formally, a *Concurrent data structure* DS is a tuple (R, I, A, P) where:

$R = (r_1, r_2, \cdots)$ is a sequence of atomic objects, $r_i = (G_{r_i}, D_{r_i}, O_{r_i})$.
$I = \{\mathbf{v^1}, \mathbf{v^2}, \cdots\}$ is a set of vectors which define the possible initial states of the atomic objects. Each vector $\mathbf{v^j} = (v_1^j, v_2^j, \cdots) \in I$ maps each atomic object $r_i \in R$ to a state $v_{r_i}^j \in D_{r_i}$.
$A = \{a_1, \cdots, a_n\}$ is a set of procedures for performing the operations of the data structure.
$P = \{p_1, p_2, \cdots\}$ is the set of processes that can access the data structure.

A *state vector* of a data structure $DS = (R, I, A, P)$ is a description of the states of the atomic objects in a given moment, and is given by a vector $\mathbf{v} = (v_1, v_2, \cdots)$, where $v_i \in D_{r_i}$ denotes the state of r_i in $\mathbf{v}$. DS has a finite memory if it has finitely many state vectors (this is equivalent, in some precise sense, to the requirement that it has finitely many objects, each of which has finitely many states).

For example, if the data structure is a queue, then A includes the *remove* procedure, and an *insert_a* procedure for each element a which can be inserted to the queue; each of these procedures consists of internal operations, and of atomic operations on the atomic objects of the underlying concurrent system.

Informally, a process executes a procedure by executing a sequence of *atomic steps*. Each such step is composed of two parts:

1. Activating an atomic operation op on an atomic object r.
2. Changing the process' state according to its current state and the value returned by the atomic operation op.

Formally, a procedure a is defined by its *operation tree* $T_a = (V_a, E_a)$. The set of vertices V_a is the set of *states* of a, where the root is the initial state of a. The set of edges E_a is the set of the atomic steps that may be executed in a. Each such edge e is identified by a 5-tuple (s, op, r, ℓ, t), denoting the process' old state s, the operation op, the atomic object r on which op was executed, the value ℓ returned by the operation, and the process' new state t;[1] the edge e above is directed from the vertex s to its son t. The out-degree of a vertex $s \in T_a$ is the number of the distinct values ℓ that can be returned by op, that is – the cardinality of $range(\lambda)$. In particular, for every possible $\ell \in range(\lambda)$ there is an edge leaving the vertex s. A leaf in the tree corresponds to a state which terminates the execution of procedure a.

A *process* p is a state machine, defined by a set of procedures $A_p \subseteq A$ which it may perform. When all the processes are identical, then $A_p = A$ for every process $p \in P$. The set of states of p is the union $\cup_{a \in A_p} V_a \cup \{idle\}$. p is in the *idle* state when it is not involved in the execution of any procedure. p can perform the following state transitions, for each procedure $a \in A_p$:

(a) *begin(a)*: Moving from the *idle* state to the initial state of a, which corresponds to the root of T_a,
(b) *end(a)*: Moving from a terminal state, which corresponds to a leaf in T_a, to the *idle* state, and
(c) Any step which corresponds to an edge in E_a.

2.2 Public Data Structures

In this work we are interested in special concurrent data structure where the set of processes performing operations on it is an infinite set of identical processes.

Definition 1. *Public data structure* $DS = (R, I, A)$ is a concurrent data structure $DS' = (R, I, A, P)$ where the set of processes P is an infinite set of identical processes, and the set of procedures that each process $p \in P$ can perform is A.

From now on when we refer to a data structure we mean a public data structure.

2.3 Runs Over Public Data Structures

Informally, a *run* x over a data structure $DS = (R, I, A)$ is a pair $(\mathbf{v}, S)$ where $\mathbf{v}$ is a state vector which denotes the contents of the shared memory at the beginning of x ($\mathbf{v}$ is not necessary in I), and S is a (finite or infinite) sequence of *events* taken by processes from P during x, where an event is a state transition of one

[1] Observe that s and ℓ define the remaining three components of an atomic step, which are added only for convenience.

of the types (a)–(c) above. An event which is not an $end(a)$ or a $begin(a)$ event will be denoted by $step(op, \delta, \lambda, r, v, v')$, where $op = (\delta, \lambda)$ is the atomic operation that was performed in the step, r is the atomic object on which op was applied, v is the state of the atomic object before the step was taken and $v' = \delta(v)$ is the state of the atomic object after the transitions function was activated (the parameters δ, λ and v' are redundant for the definition, but convenient to use). Note the difference between step in a run and step of a procedure, defined in Section 2.1.

At the beginning of a run $x = (\mathbf{v}, S)$, all the processes in P are in the *idle* state, and the sequence of events S consists of events taken by various processes, according to the procedures which define them. A process p excecutes a procedure a in a run by following a path in the operation tree T_a which defines a; this path always begins at the root, and each atomic step of p corresponds to an edge e leaving the current vertex in T_a to one of its sons, where e is determined by the value returned by atomic operation executed by p. When the sequence of events S is finite, we say that the run is finite. The state vector $\mathbf{v}$ is called $initial(x)$, and when the run is finite, the state vector $\mathbf{u}$ which represents the contents of the shared memory at the end of x is called $final(x)$. We say that x is *a run from* $\mathbf{v}$ *to* $\mathbf{u}$ if $initial(x) = \mathbf{v}$ and $final(x) = \mathbf{u}$.

Let $x = (\mathbf{v}, S)$ and $y = (\mathbf{v}, S')$ be two runs with the same initial vector. The run y is a prefix of x (and x is an extension of y) if S' is a prefix of S. The run x is denoted as the concatenation $x = y \cdot z$, where $z = (x - y)$ is the suffix of S obtained by removing S' from S. The sequence of events z is denoted as a *partial run.* $(\mathbf{v}, null)$, where *null* is an empty sequence, denotes an *empty run.* A run x is a *legal* run of a data structure if $\mathbf{v}$ belongs to I.

A process p is *involved in a run* x if it is not idle in x (i.e., if it has started some procedure $a \in A$ but has not completed it yet).

Through this work we use the following notation, that defines the next operation a given process is going to take in a given run.

Definition 2. A process p is $(\delta, r)-loaded$ in a run x if it is in a state in which it is going to perform the operation $op = (\delta, \lambda)$ on the atomic object $r = (G, D, O)$, for some $op \in G$.

2.4 Wait-Free Public Data Structures

We require from any public data structure that it satisfies the *wait-freedom* property, which is: if a process p is activated infinitely often in a legal run x, then p completes the execution of any procedure that it starts during x.

As we show later, there are wait-free public data structures in which there is no upper bound on the number of steps that a process may execute in order to complete the execution of a given procedure. A data structure which does not have this unpleasant property is called *bounded* wait-free data structure.

Definition 3. A data structure $DS = (R, I, A)$ is *bounded wait-free* if there exists a constant $\mathtt{h}$ such that in any run x over DS, every procedure is completed within at most $\mathtt{h}$ steps.

A wait-free data structure which is not bounded is called an *unbounded* wait-free data structure.

3 An Example of an Unbounded Wait-free Data Structure

In this section we present an example of an unbounded wait-free data structure, denoted DS_{ub}, which uses only two binary atomic objects. $DS_{ub} = (R, I, A)$ where:

- R consist of two identical atomic objects, r_1 and r_2, defined by the triple (G, D, O) where:
  The sets of states and output values are given by $D = O = \{0, 1\}$.
  The atomic operations are $G = \{read, set\_1, inc\}$ where:
  $inc = (\delta_{inc}, \lambda_{inc})$, where $\delta_{inc}(v) = v + 1 (\text{mod } 2)$ and $\lambda_{inc}(v) = v$.
  $read = (\delta_{read}, \lambda_{read})$, where $\delta_{read}(v) = v$ and $\lambda_{read}(v) = v$.
  $set\_1 = (\delta_{set\_1}, \lambda_{set\_1})$, where $\delta_{set\_1}(v) = 1$ and $\lambda_{set\_1}(v) = v$.
- $I = \{(0, 0)\}$ (That is, there is a single initial state vector in which both objects are in state 0.)
- $A = \{a_f\}$, where the procedure a_f is described in figure 1 in a Pascal like language (the translation of the procedure to an operation tree is immediate).

Informally, a process executes the procedure a_f as follows: it starts by reading r_1; if the value read is 0, then it increments r_2 by 1 (mod 2), writes 1 in r_1, and terminates. Otherwise (i.e., the value read in r_1 is 1), it reads r_2 repeatedly, and it terminates after it reads the same value in two successive read operations.

In order to prove the properties of the above data structure, we observe the following:

1. A process that reads 0 in r_1 terminates the execution of procedure a_f in three steps: reading r_1, incrementing r_2, and writing 1 in r_1.
2. Only processes that read 0 in r_1 increments r_2.
3. Once the value of r_1 is changed to 1, it remains so forever. Hence, in each run x in which some process writes 1 in r_1, there is a fixed number of processes, M_x (M_x depends on x), which read 0 in r_1.

Lemma 3.1 *The data structure DS_{ub} defined above is a wait-free but not bounded wait-free data structure.*

Proof. First we prove that the data structure DS_{ub} is a wait-free and then we prove that it is not bounded wait-free.

To show that DS_{ub} is wait-free, we must show that in each run x, each execution of a_f terminates. Consider such an execution of a_f by a process p, which does not terminate within three steps. Then by 1 above, p has read 1 in r_1, and hence p starts reading r_2 repeatedly, and it terminates when it gets identical values in two successive read operations.

By 3, only M_x processes read 0 in r_1 for some constant M_x, and hence by 2 the value of r_2 is incremented at most M_x times in the run x. The proof is completed by observing that, between any two successive reads of r_2 by p (except for the last one), r_2 must have been incremented - and hence p reads r_2 at most $M_x + 2$ times in x, and thus p must terminate after at most $M_x + 3$ steps.

Next we prove that DS_{ub} is not bounded wait-free data structure, by showing that for every given L, there is a run x_L in which some process p take at least L steps in order to complete the execution of a_f.

Let $H = \{q_1, \cdots, q_L\}$ be a set of L idle processes, $p \notin H$. We construct the run x_L in the following way:

```
1: begin
2:     c1 := r1;                           /* read-operation r1*/
3:     if c1 = 0
4:         r2 := r2 + 1(mod2);             /* inc-operation r2 */
5:         r1 := 1;                        /* write operation r1 */
6:     else
7:         c1 := r2;                       /* read-operation r2 */
8:         c2 := r2;                       /* read-operation r2 */
9:         while (c1 ≠ c2)                 /* any change? */
10:            c1 := c2;
11:            c2 := r2;                   /* read-operation r2 */
12:        end;
13:    end;
14: end;
```

Figure 1: The procedure a_f

1. Initially, each of the processes in H reads 0 in r_1.
2. Then, the process q_1 completes performing its procedure (and in particular, it writes 1 in r_1).
3. Next, process p reads 1 in r_1, and then it reads the value of r_2 (in line 7).
4. Now, for $i = 1 \cdots L$, process q_i increments r_2, and then process p reads r_2 (observe that in any two successive read operations of r_2, p will always read two different values).

Since the number of processes in H is L, the process p takes $L + 2$ atomic steps in x_L before it completes executing a_f. □

In the next sections we will define and study a class of public data structures in which wait-freedom is equivalent to bounded wait-freedom. In particular, we define *periodic* data structures, and prove that a periodic wait-free data structure with finite memory must be bounded wait-free.

4 Periodic Data Structures

Let δ be a transition function and v a state. For $n \geq 0$, $\delta^n(v)$ is defined by: $\delta^0(v) = v$, and for $n > 0, \delta^n(v) = \delta(\delta^{n-1}(v))$.

Definition 4. Let δ be a transition function and let v be a state. Then $order(\delta, v)$, the *order of δ on v*, is the minimum positive integer d s.t. $\delta^d(v) = v$. If for each $d > 0, \delta^d(v) \neq v$ then $order(\delta, v) = \infty$. For an operation $op = (\delta, \lambda)$, $order(op) = \max_v\{order(\delta, v)\}$.

Examples: In the data structure DS_{ub}, the order of the *read* operation is 1, the order of the *inc* operation 2, and the order of the *set_1* operation is ∞. The order of an *increment* operation in a b-valued atomic counter, which increments the value of the counter by $1(\bmod b)$, is b, and the order of the *read* operation in such a counter is 1.

Definition 5. Let $r = (G, D, O)$ be an atomic object. Then r is *periodic* if for each $v \in D$ and $op = (\delta, \lambda) \in G$, $order(\delta, v)$ is finite.

Examples: A b-valued atomic counter is periodic, while the atomic objects in the data structure DS_{ub} are not, since the order of *set_1* is ∞.

Definition 6. A data structure $DS = (R, I, A)$ is *periodic* if all the atomic objects in R are periodic.

The following theorem is the main result of this paper.

Theorem 4.1 *Let $DS = (R, I, A)$ be a wait-free data structure. If DS is periodic and has finite memory, then DS is bounded wait-free.*

The above theorem implies that any wait-free data structure which uses only a finite number of atomic counters (like counting networks and their variants) must be bounded wait-free. It also implies that the use of the atomic operation *set_1* in the data structure DS_{ub} is essential for making it an unbounded wait-free data structure, since it is the only operation in that data structure whose order is ∞.

5 The Reconstruction Property

As a first and main step towards the proof of Theorem 4.1, we prove in this section that a periodic data structure satisfies the *reconstruction property*; informally, this property guarantees that if there is a run x over DS which starts when the state vector is $\mathbf{v}$ and ends when the state vector is $\mathbf{u}$, then there is another run, x', which starts when the state vector is $\mathbf{v}$, reaches an intermediate state in which the state vector is $\mathbf{u}$, and terminates when the state vector is again $\mathbf{v}$.

Definition 7. A data structure DS satisfies the *reconstruction property* if the following holds for any pair of vectors $\mathbf{v}$ and $\mathbf{u}$:
If there is a run x where $initial(x) = \mathbf{v}$ and $final(x) = \mathbf{u}$, then there is a run $x' = y \cdot z$, such that $initial(x') = initial(y) = \mathbf{v}$, $final(y) = \mathbf{u}$, and $final(x') = final(y \cdot z) = \mathbf{v}$.

Note that the data structure DS_{ub} does not satisfy the reconstruction property: there is a run x over DS_{ub} from the vector $\mathbf{v} = (0,0)$ to $\mathbf{u} = (1,0)$, but there is no run x' from vector $\mathbf{v}$ to itself through $\mathbf{u}$, since there is no operation that changes the state of r_1 from 1 to 0.

Lemma 5.1 (The Reconstruction Lemma) *Let $DS = (R, I, A)$ be a periodic data structure. Then DS satisfies the reconstruction property.*

Proof. Let x be a given run from $\mathbf{v}$ to $\mathbf{u}$. We have to prove that there is a run $x' = y \cdot z$, such that $initial(x') = initial(y) = \mathbf{v}$, $final(y) = \mathbf{u}$, and $final(x') = final(z) = \mathbf{v}$. This is trivial if $\mathbf{v} = \mathbf{u}$, so assume that $\mathbf{v} \neq \mathbf{u}$.

Let $x = x_1 x_2 \cdots x_L$, where x_i is (a partial run consisting of) the i-th event in x. We construct the run $x' = y \cdot z$, where:

1. y is a run from $\mathbf{v}$ to $\mathbf{u}$ defined by the concatenation $y = y_1 \cdots y_L$, such that $initial(y_i) = initial(x_i)$ and $final(y_i) = final(x_i)$.
2. z is a partial run from $\mathbf{u}$ to $\mathbf{v}$ defined by $z = z_L \cdots z_1$, where $initial(z_i) = final(x_i)$ and $final(z_i) = initial(x_i)$,

as described in Figure 2.

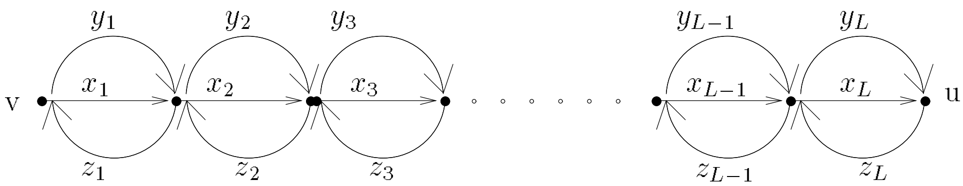

Figure 2: Construction of the run $x' = y \cdot z$

Let $P = \{p_1, \cdots, p_\ell\}$ be the set of processes which are involved in the run x, and let $\mathcal{D} = max\{order(\delta, v)|$ the operation $op = (\delta, \lambda)$ is applied on state v during x by some process $p \in P\}$ ($\mathcal{D}$ is finite since the data structure DS is periodic and x is a finite run).

The run y is constructed by induction in the following way: at stage i, $1 < i \leq L$, we extend the run $y_1 \cdots y_{i-1}$ which was constructed in the previous stage to the run $y_1 \cdots y_i$. For $i = 1, \cdots, L$, each of the runs $y_1 \cdots y_i$ satisfies the following requirements:

1. $initial(x_i) = initial(y_i)$ and $final(x_i) = final(y_i)$.
2. For each process $q \in P$ there is a set of processes H_q^i, such that at the end of y_i, all the processes in H_q^i are in the same state as q in the end of run x_i, and $\mathcal{D}(\mathcal{D}+1)^{L-i}$ divides $|H_q^i|$.
3. If x_i contains an event in which some process performs the atomic step $step(op, \delta, \lambda, r, v, v')$, then at the end of y_i there is a set U^i of at least $\mathcal{D}$ processes, which are $(\delta, r) - loaded$.

Base of induction: We construct the run y_1 and the corresponding set of processes U^1 and H_q^1 for each $q \in P$.

U^1 is the empty set, and for each $q \in P$, H_q^1 is a set of $\mathcal{D}(\mathcal{D}+1)^{L-1}$ processes which are initially in the idle state.

Let p be the process which performed the event in the run x_1. Since it is the first event in x, it is an event of the type $begin(a)$, for some procedure $a \in A$. The run y_1 is defined as follows: $initial(y_1) = \mathbf{v}$, each process from H_p^1 performs the event $begin(a)$ in y_1, and no other events are taken in y_1.

It is easily verified that (a) $initial(x_1) = initial(y_1) = final(x_1) = final(y_1)$, and (b) for each $q \in P$, at the end of y_1 all the processes in H_q^1 are at the same state as process q at the end of x_1. Thus the induction hypothesis holds for $i = 1$.

Induction step: Suppose we have constructed the partial run y_{i-1}, and for each $q \in P$ the sets H_q^{i-1}, such that conditions 1-3 hold for $i-1$. We now describe the construction for i.

Let p be the process which performs the event in the partial run x_i. Then for each process $q \neq p$ in P, we set $H_q^i = H_q^{i-1}$. It remains to define the sets H_p^i and U^i, and the partial run y_i. We distinguish between two cases:

Case 1: The event in x_i is a $begin(a)$ or an $end(a)$ event, for some procedure $a \in A$. In this case we let $H_p^i = H_p^{i-1}$ and $U^i = \emptyset$. The partial run y_i is constructed by letting each of the processes in H_p^i perform its next step, which is the same event that p performs in x_i (i.e., a $begin(a)$ or an $end(a)$ event). For each process $q \neq p$, the set H_q^i satisfies the induction hypothesis since H_q^{i-1} satisfies it for $i-1$, and since q does not take a move in x_i and no process in H_q^i takes a move in y_i. Each process in H_p^i takes in y_i the same step that p takes in x_i, and hence the induction hypothesis holds for H_p^i too. Finally, condition 1 holds for i since it holds for $i-1$ and no atomic object was modified during the partial runs x_i and y_i.

Case 2: The event in the partial run x_i is an atomic step $step(op, \delta, \lambda, r, v, v')$, taken by p. This means that at the end of the partial run x_{i-1} the process p is (δ, r)-loaded, and hence, by the induction, each process in H_p^{i-1} is (δ, r)-loaded at the end of y_{i-1}. The sets U^i and H_p^i are defined as follows: Partition the set H_p^{i-1} into $\mathcal{D}+1$ sets of equal size, $G_1^i, \cdots, G_{\mathcal{D}}^i$ and U^i. Thus, $|U^i| = |G_j^i| = \frac{|H_p^{i-1}|}{\mathcal{D}+1}, 1 \leq j \leq \mathcal{D}$, and by the induction we have that $\mathcal{D}(\mathcal{D}+1)^{L-i+1}$ divides $|U^i|$ and $|G_j^i|$ $(1 \leq j \leq \mathcal{D})$. The set H_p^i is the set G_1^i.

By the above, each process in U^i is (δ, r)-loaded at the beginning of y_i, and $|U^i| \geq \mathcal{D}$. Thus, U^i satisfies the induction hypothesis, provided that no process in U^i takes a move in y_i.

To this end, we define the partial run y_i. Let $d = order(\delta, v)$. The run y_i involves only the processes from the sets $G_1^i, \cdots, G_d^i$, in such a way that each process from the set $H_p^i = G_1^i$ performs the event $step(op, \delta, \lambda, r, v, v')$ – the same event that the process p performs in the partial run x_i.

y_i is constructed in a cyclic way by activating processes from the sets $G_1^i, \cdots, G_d^i$ in the following way: activate a process form G_1^i so that it performs its next step, which by the induction hypothesis is $step(op, \delta, \lambda, r, v, v')$; then activate in a similar manner a process from G_2^i, and so on till G_d^i; this procedure is repeated $|G_1^i| - 1$ times. Note that the above construction guarantees that each process in $H_p^i = G_i^1$ except one takes in y_i the atomic step $step(op, \delta, \lambda, r, v, v')$, which is the same step taken by p in x_i. y_i is completed by letting the only process in G_1^i that was not activated yet take its next step, which is also $step(op, \delta, \lambda, r, v, v')$.

We now show that the induction requirements are fulfilleded. As mentioned above, the set U^i satisfies the induction requirements. It is clear from the construction that the processes in H_q^i for each $q \in P$ are in the same state as q in the end of run x_i and $\mathcal{D}(\mathcal{D}+1)^{L-i}$ divides $|H_q^i|$. Thus conditions 2 and 3 hold. It remains to show that 1 holds too. The induction assumption for $i-1$ implies that $initial(y_i) = initial(x_i)$, so it remains to show that $final(x_i) = final(y_i)$. In the partial run x_i only the event $step(op, \delta, \lambda, r, v, v')$ was performed, thus the state of r was changed from v to $v' = \delta(v)$, and the states of all other objects remain unchanged. In the partial run y_i the function δ was applied $Kd + 1$ times on r, for $K = |H_p^i| - 1$. Thus the state of r was changed to $\delta^{kd+1}(v) = \delta(v)$, and the state of no other object was changed. We conclude that $final(x_i) = final(y_i)$.

At the end of run y_L we get $final(y) = final(y_L) = final(x_L) = \mathbf{u}$.

We now complete the proof of the lemma by constructing the partial runs z_i described at the beginning of this proof. For each $L \geq i \geq 1$, only processes from U^i are activated in z_i.

In the construction of the partial run $z_i, L \geq i \geq 1$ we distinguish between two cases according to the type of the event in the partial run x_i. In the first case the event is a $begin(a)$ or an $end(a)$ event, which does not change the state of any object. In that case z_i is an empty run and it is clear that $initial(z_i) = final(x_i)$ and $final(z_i) = initial(x_i)$.
In the second case the event in the partial run x_i is an atomic step $step(op, \delta, \lambda, r, v, v')$. According to the construction of y_i, there is a set U^i of at least $\mathcal{D}$ processes which are (δ, r)-loaded. Let $d = order(\delta, v)$. The partial run z_i is constructed by letting $d-1$ processes from the set U^i perform their next atomic step (note that U^i contains at least $\mathcal{D} \geq d$ processes). Since each process in U^i is (δ, r)-loaded, we get that at the end of z_i the state of r is $\delta^{d-1}(v') = v$, and hence $final(z_i) = initial(x_i)$, as claimed.

To conclude, we have that the partial run $z = z_L \cdots z_1$ satisfies $initial(z) = initial(z_L) = \mathbf{u}$ and $final(z) = final(z_1) = \mathbf{v}$, which completes the proof. $\square$

6 Proof of the Main Theorem

Let $a \in A$ be a given procedure of a data structure $DS = (R, I, A)$. If $height(T_a)$, the height of the tree T_a, is h_a, then each execution of a terminates within h_a steps. In particular, if for every procedure $a \in A$ it holds that $height(T_a) < \mathtt{h}$ for some constant $\mathtt{h}$, then DS is bounded wait-free. In this section we show that a slightly weaker property than the one described above holds for any wait-free periodic data structure with finite memory. Later we use this result to prove that every such data structure must be bounded wait-free, which completes the proof of Theorem 4.1. First we need some definitions.

Let $\mathbf{v}$ be a given state vector. $V_{reach}(\mathbf{v})$ is the set of state vectors which are *reachable* from $\mathbf{v}$ in DS:

$$V_{reach}(\mathbf{v}) = \{\mathbf{u} \mid \text{ there is a run from } \mathbf{v} \text{ to } \mathbf{u}\}.$$

For each atomic object $r = (G_r, D_r, O_r)$, $D_r(\mathbf{v})$ is the set of states which r gets in $V_{reach}(\mathbf{v})$:

$$D_r(\mathbf{v}) = \{c \mid \exists \mathbf{u} \in V_{reach}(\mathbf{v}), \text{ the state of } r \text{ in } \mathbf{u} \text{ is } c\}.$$

Finally, for each atomic operation $op = (\delta, \lambda) \in G_r$, $O_r(\mathbf{v}, op)$ is the set of values returned by applying op when the state of r is in $D_r(\mathbf{v})$:

$$O_r(\mathbf{v}, op) = \{\ell \mid \exists c \in D_r(\mathbf{v}), \lambda(c) = \ell\}.$$

It is clear that for each $\mathbf{v}$ it holds that $D_r(\mathbf{v}) \subseteq D_r$, and for each op, $O_r(\mathbf{v}, op) \subseteq O_r$.

Let a be a procedure and $\mathbf{v}$ be a state vector. $T_a(\mathbf{v})$, the *operation tree of the procedure a induced by* $\mathbf{v}$, is a sub-tree of the operation tree T_a, which does not include state transitions which result in output values ℓ which are never achievable in runs that start at $\mathbf{v}$. Formally, $T_a(\mathbf{v})$ is defined as follows:
Initially let $T_a(\mathbf{v}) = T_a$. Then scan the vertices of $T_a(\mathbf{v})$ ordered according to their height (i.e., the root is first, then all the sons of the root and so on[2]). Upon scanning a non-leaf vertex s do the following for each son t of s: Let (s, op, r, ℓ, t) be the step defining the state transition from s to t. Then if $\ell \notin O_r(\mathbf{v}, op)$, delete from $T_a(\mathbf{v})$ the vertex t and all its descendants.

The *legal operation forest* of procedure a in a data structure $DS = (R, I, A)$, denoted F_a, is defined as the union of all the operation trees induced by initial state vectors:

$$F_a = \cup_{\mathbf{v} \in I} T_a(\mathbf{v})$$

For a forest F, let $height(F)$ denote the maximal height of the trees in F ($height(F) = \infty$ if there is no such maximal height). In order to prove Theorem 4.1 it is sufficient to show that there exists a constant $\mathtt{h}$ such that for every $a \in A$, it holds that $height(F_a) < \mathtt{h}$. Since A is finite and for each $a \in A$ the number of trees in F_a is finite, such an $\mathtt{h}$ exists if for each $a \in A$ and $\mathbf{v} \in I$, the height of $T_a(\mathbf{v})$ is finite (note that the height of $T_{a_f}((0,0))$, the operation tree of the procedure a_f of the data structure DS_{ub} induced by the initial vector (0,0), is ∞). The main step in proving this last claim is the following lemma.

Lemma 6.1 *Let $DS = (R, I, A)$ be a periodic wait-free data structure and let $\mathbf{v}$ be an initial state vector of DS. Then for each $a \in A$, the operation tree $T_a(\mathbf{v})$ does not contain an infinite path.*

Proof. We show that if $T_a(\mathbf{v})$ contains an infinite path then there is a legal run x over DS in which some process p takes an infinite number of steps during a single execution of the procedure a, which contradicts the assumption that DS is wait-free.

Let $(s_1, s_2, \cdots)$ be an infinite path in the tree $T_a(\mathbf{v})$, where s_1 is the root of the tree $T_a(\mathbf{v})$. For $i = 1, 2, \cdots$, let $e_i = (s_i, op_i, r_i, \ell_i, s_{i+1})$ be the atomic step defined by the edge (s_i, s_{i+1}), which corresponds to the transition from s_i to s_{i+1}.

The infinite legal run x is defined as the limit of a sequence of finite runs $(x_1, x_2, \cdots)$, where x_{i+1} is the extension of the run x_i, and each run x_i satisfies the follows:

[2] Note that since the DS has a finite memory, every vertex in T_a has only finitely many sons, and hence this is a total ordering of the vertices of T_a. It is possible to define such an ordering also when vertices of T_a may have an infinite out-degree.

1. During the run x_i the process p executes the atomic steps $e_1, e_2, \cdots, e_i$.
2. $initial(x_i) = final(x_i) = \mathbf{v}$ (i.e., x_i is a run from $\mathbf{v}$ to itself).

First we construct the run x_1, in which p performs step $e_1 = (s_1, op_1, r_1, \ell_1, s_2)$: By the definition of $T_a(\mathbf{v})$, there is a vector $\mathbf{u}^1 \in V_{reach}(\mathbf{v})$ such that the state of r_1 in $\mathbf{u}^1$ is c_1, for some $c_1 \in D_{r_1}(\mathbf{v})$ satisfying $\lambda(c_1) = \ell_1$. In particular, if we let p start executing procedure a, and then take its first step in a when the state vector is $\mathbf{u}^1$, then it will perform the step e_1. Since $\mathbf{u}^1 \in V_{reach}(\mathbf{v})$ there is a run from $\mathbf{v}$ to $\mathbf{u}^1$. By Lemma 5.1 there is a run $w^1 = w_1^1 \cdot w_2^1$ from $\mathbf{v}$ to itself such that $final(w_1^1) = \mathbf{u}^1$.

The run x_1 is constructed by a concatenation of the runs $x_1 = w_1^1 \cdot z_1 \cdot w_2^1$ where:

1. The run w_1^1 as defined above satisfies $initial(w_1^1) = \mathbf{v}, final(w_1^1) = \mathbf{u}^1$.
2. Let $op_1 = (\delta_1, \lambda_1)$, and let $d_1 = order(\delta_1, c_1)$. The partial run z_1 from $\mathbf{u}^1$ to itself is constructed by activating the process p so that it will take its first atomic step in a, and then activating $d_1 - 1$ idle processes so that each of them also takes its first atomic step in a. Since (a) the first atomic step in a performs the atomic operation $op_1 = (\delta_1, \lambda_1)$ on r_1, (b) the state of r_1 in $\mathbf{u}^1$ is c_1, and (c) $d_1 = order(\delta_1, c_1)$, it follows that $initial(z_1) = final(z_1) = \mathbf{u}^1$.
3. The run w_1^2 as defined above satisfies $initial(w_1^2) = \mathbf{u}^1, final(w_1^2) = \mathbf{v}$.

Assume that we have constructed a run x_{i-1} satisfying the assumptions ($i > 1$). Now we show how to extend x_{i-1} to the run x_i, in which p performs the step $e_i = (s_i, op_i, r_i, \ell_i, s_{i+1})$.

By the definition of $T_a(\mathbf{v})$, there is a vector $\mathbf{u}^i \in V_{reach}(\mathbf{v})$ such that the contents of r_i in $\mathbf{u}^i$ is c_i, for some $c_i \in D_{r_i}(\mathbf{v})$ satisfying $\lambda(c_i) = \ell_i$.

Since $\mathbf{u}^i \in V_{reach}(\mathbf{v})$, Lemma 5.1 implies that there is a run $w^i = w_1^i \cdot w_2^i$ from $\mathbf{v}$ to itself where $final(w_1^i) = \mathbf{u}^i$. The run x_i is constructed as a concatenation of the runs $x_i = x_{i-1} \cdot y_i \cdot w_1^i \cdot z_i \cdot w_2^i$ where w_1^i and w_2^i are as defined above, and y_i and z_i are defined below:

1. Let $op_i = (\delta_i, \lambda_i)$, and let $d_i = order(\delta_i, c_i)$. The run y_i is a concatenation of $d_i - 1$ copies of the run x_{i-1}, where the sets of processes involved in each such copy are mutually disjoint, and are also disjoint from the sets of processes involved in w^i. Therefore $final(y_i) = initial(y_i) = \mathbf{v}$, and at the end of the run y_i there is a set H_i of at least $d_i - 1$ processes whose state at the end of y_i is s_i (if $d = 1$ then y_i is an empty run).
2. z_i is a partial run from $\mathbf{u}^i$ to itself, constructed as follows: First we activate process p so it takes its next step, and then we activate $d_i - 1$ processes from the set H_i, such that each of them takes its next step, in which it executes the atomic operation $op_i = (\delta_i, \lambda_i)$ on r_i. Since $d_i = order(\delta_i, c_i)$, the partial run z_i satisfies $final(z_i) = intial(z_i) = \mathbf{u}^i$.

It is easily verified that $final(y_i) = \mathbf{v}$, and at the end of y_i the state of p is s_{i+1}, thus the induction assumptions holds for i. This completes the proof of the lemma. □

Note that the proofs of lemmas 5.1 and 6.1 are valid also in the case that DS is a periodic data structure which has an infinite memory.

Corollary 6.1 *Let $DS = (R, I, A)$ be a data structure satisfying the assumptions of Lemma 6.1. Then for each $a \in A$ and for each $\mathbf{v} \in I$, the height of $T_a(\mathbf{v})$ is finite.*

Proof. Let such a and $\mathbf{v}$ be given. Since DS has a finite memory, R is finite, and for each object $r = (G_r, D_r, O_r) \in R$ the set O_r is finite. Since the out-degree of each vertex in T_a is bounded by $\max_{r \in R}\{|O_r|\}$, we have that the out-degree of each vertex in T_a is finite. Since $T_a(\mathbf{v})$ is a sub-tree of T_a, the same holds also for $T_a(\mathbf{v})$.

The proof is completed by noting that if the height of $T_a(\mathbf{v})$ is not finite, then $T_a(\mathbf{v})$ has infinitely many vertices, which implies by the Infinity Lemma [Kön36] that $T_a(\mathbf{v})$ contains an infinite path - but this contradicts Lemma 6.1. □

Now we complete the proof of Theorem 4.1: By Corollary 6.1, for each $a \in A$ and $\mathbf{v} \in I$, $height(T_a(\mathbf{v}))$ is finite. Since both A and I are finite, it means that there is a constant $\mathtt{h}$ such that for each such a and $\mathbf{v}$, it holds that $height(T_a(\mathbf{v})) < \mathtt{h}$, hence each execution of a in a legal run of DS terminates within at most $\mathtt{h}$ steps, and thus DS is bounded wait-free. □

7 A Generalization of the Main Theorem

In this section we first generalize theorem 4.1, by showing that the requirement of finite memory can be replaced by two weaker requirements, and then we show by counter examples that neither of these two requirements can be eliminated.

Definition 8. Let $DS = (R, I, A)$ be a data structure. DS satisfies the *finite output property* if for each $a \in A$ and $\mathbf{v} \in I$, every vertex in $T_a(\mathbf{v})$ has finite out-degree.

Example: If for each $r \in R$, the set of output values O_r is finite, then DS satisfies the finite output property.

Theorem 7.1 *Let $DS = (R, I, A)$ be a wait-free public data structure. If DS is periodic, I is finite, and DS satisfies the finite output property, then DS is bounded wait-free.*

Note that every periodic data structure which has a finite memory satisfies the assumption of Theorem 7.1, but the converse is not true.

Theorem 7.1 is proved along the same outline as the proof of Theorem 4.1, by noting that the proof of Lemma 6.1 requires only that DS is periodic and satisfies the finite output property, and the proof of Corollary 6.1 requires, in addition to these two requirements, only that I is finite.

The example in Section 3 shows that being periodic is essential for a data structure to satisfy Theorem 7.1. Next we show that each of the other two properties are essential too, by showing that if for a given periodic data structure $DS = (R, I, A)$, either I is infinite, or DS does not satisfy the finite output property, then wait-freedom does not necessarily imply bounded wait-freedom.

The first example is a data structure $DS_a = (R_a, I_a, A_a)$ which satisfies the finite output property, but $|I_a| = \infty$. The second example is a data structure $DS_b = (R_b, I_b, A_b)$ in which I_b is finite (in fact $|I_b| = 1$), but the finite output property is not satisfied.

DS_a is given by:

- $R_a = (r_1, r_2, \cdots)$ consists of an infinite number of identical atomic objects, defined by the triple (G, D, O), where $D = O = \{0, 1\}$, and G consists of a single operation - the *read* operation defined earlier.

- I_a consists of infinitely many initial vectors $I_a = \{\mathbf{v}_i \mid i > 0\}$ where in $\mathbf{v}_i$ the states of all the atomic objects $r_j, j \neq i$ are 0 and the state of the atomic object r_i is 1.
- A_a consists of a single procedure, which does the following: For $i = 1, 2, \cdots$, it reads r_i until it reads the value 1 and stops.

It is obvious that when the initial vector is $\mathbf{v}_i$, each execution of the procedure will terminate after i steps. Hence DS_a is an unbounded wait-free data structure.

The data structure $DS_b = (R_b, I_b, A_b)$ is defined as follows:

- R_b consists of a single atomic object $r = (G, D, O)$, where D and O are the set of the natural numbers, and G consists of three operations:
  1. The *read* operation, which returns the state of the object r.
  2. The inc_0 operation, which increments the state of r if it is an even integer, and decrements it if it is an odd integer.
  3. The inc_1 operation, which increments the state of r if it is an odd integer, and decrements it if it is an even integer.

  Note that $order(inc_0) = order(inc_1) = 2$, hence DS_b is a periodic data structure.
- I_b consist of a single vector which initializes r to 0.
- $A_b = \{a_0, a_1, mread\}$, where:
  The procedure a_0 applies the operation inc_0 on the atomic object r and stops.
  The procedure a_1 applies the operation inc_1 on the atomic object r and stops.
  The procedure *mread* reads the value of r, and if the value read is c, it repeats reading it another c times and stops.

It is obvious that DS_b is wait-free since the number of steps required to complete any execution of a procedure in A_b is finite.

In order to show that DS_b is not bounded wait-free, we show that for each integer n there is a run x_n, at the end of which the state of r is 2n. This is sufficient, since at the end of such a run x_n, $2n$ steps are needed for performing the procedure *mread.* Such a legal run x_n is obtained by activating n processes in a sequential order, letting each of them perform the procedure a_0 first, then perform the procedure a_1 and stop.

8 Conclusion and Further Research

We introduced the concept of public data structure, and invetsigated the relation between wait-freedom and bounded wait-freedom in such structures. In particular, we have shown that in data structures which use only periodic objects, wait-freedom is equivalent to bounded wait-freedom.

While certain concurrent objects are periodic, there are common, simple objects which are not periodic (e.g., read/write registers). We have used such objects to show that, in public data structures, wait-freedom is not equivalent to bounded wait-freedom. For comparison, this is not the case in finite data structures which are accessible to a bounded number of processes, each of which may excecute a bounded number of procedures (non-atomic operations) in each run; i.e., in this latter case wait-freedom is equivalent to bounded wait-freedom.

Public data structures allow access to infinitely many processes. each of which may perform arbitrarily many non-atomic operations in each execution. It is interesting to know under what conditions wait-freedom implies bounded wait-freedom in data structures which are accessible by a finite number of processes, each of which may execute arbitrarily many procedures in each run.

References

[AA92] E. Aharonson and H. Attiya. Counting networks with arbitary fan-out. *ACM-SIAM Symp on Discrete Algorithms*, pages 104–113, 1992.

[AH91] R.J. Anderson and H.Woll. Wait-free parallel algorithms for the union-find problem. In *Proc. 23rd ACM Symp. on Theory of Computing*, pages 370–380, 1991.

[AHS91] J. Aspnes, M. Herlihy, and N. Shavit. Counting networks and multiprocessor coordination. In *Proc. 23rd ACM Symp. on Theory of Computing*, pages 348–358, May 1991.

[AT93] 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*, December 1993.

[BMT95] H. Brit, S. Moran, and G. Taubenfeld. Public data structures: Counters as a special case. In *Proc. 3rd Israel Symposium on Theory of Computing and Systems*, January 1995.

[BS77] R. Bayer and M. Schkolnick. Concurrency operations on B-trees. *Acta Informatica*, 9(2):1–21, 1977.

[CIL87] B. Chor, A. Israeli, and M. Li. On processor coordination using asynchronous hardware. In *Proc. 6th ACM Symp. on Principles of Distributed Computing*, pages 86–97, 1987.

[Ell80a] C.S. Ellis. Concurrency search and insert in 2-3 trees. *Acta Informatica*, 14:63–86, 1980.

[Ell80b] C.S. Ellis. Concurrent search and insertion in AVL trees. *IEEE Transactions on Computers*, c-29:811–817, 1980.

[Ell85] C.S. Ellis. Distributed data structures: A case study. *IEEE Transactions on Computers*, c-34(12):1178–1185, 1985.

[FLP85] 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.

[Her90] P. M. Herlihy. A methodology for implementing highly concurrent data structures. In *Proc. 2nd ACM Symp. on Principles and Practice of Parallel Programming*, pages 197–206, 1990.

[Her91a] M. Herlihy. Impossibility results for asynchronous PRAM. In *Proc. 3rd ACM Symp. on Algorithms and Architectures*, pages 327–336, 1991.

[Her91b] M. Herlihy. Wait-free synchronization. *ACM Trans. on Programming Languages and Systems*, 11(1):124–149, January 1991.

[HLS92] M. Herlihy, B. Lim, and N. Shavit. Low contention load balancing on large-scale multiprocessors. In *Proceedings of the 3rd Annual ACM Symposium on Parallel Algorithms and Architectures*, July 1992.

[HSW91] M. Herlihy, N. Shavit, and O. Waarts. Low contention linearizable counting. In *Proc. 32nd IEEE Symp. on Foundations of Computer Science*, pages 526–535, October 1991.

[HU79] J. E. Hopcroft and J. D. Ullman. *Introduction to Automata Theory, Languages and Computations*. Addison-Wesley, 1979.

[HW87] M. Herlihy and J. Wing. Axioms for concurrent objects. In *Proc. 14th ACM Symp. on Principles of Programming Languages*, pages 13–26, 1987.

[IR93] A. Israeli and L. Rappoport. Efficient wait-free implementaion of a concurrent priority queue. In *Proceedings of 7-th International Workshop on Distributed Algorithms*, pages 1–17, November 1993.

[Kön36] D. König. Theorie der endlichen und unendlichen graphen. Leipzig, 1936. reprinted by Chelsea, 1950.

[KP92] M. Klugerman and C. Plaxton. Small-depth counting networks. In *Proc. 24rd ACM Symp. on Theory of Computing*, pages 417–428, October 1992.

[Lam83] L. Lamport. Specifying concurrent program modules. *ACM Trans. on Programming Languages and Systems*, 5(2):190–222, 1983.

[Lam86a] L. Lamport. The mutual exclusion problem: Statement and solutions. *Journal of the ACM*, 33:327–348, 1986.

[Lam86b] L. Lamport. On interprocess communication, parts I and II. *Distributed Computing*, 1(2):77–101, 1986.

[LY81] P.L. Lehman and S.B. Yao. Efficient locking for concurrent operations on B-trees. *ACM Transactions on Database Systems*, 6(4):650–670, 1981.

[MT93] S. Moran and G. Taubenfeld. Lower bound on wait-free counting. In *Proc. 12th ACM Symp. on Principles of Distributed Computing*, August 1993.

[MTY95] S. Moran, G. Taubenfeld, and I. Yadin. Concurrent counting. *Journal of Computer and System Science*, 1995. to Appear, preliminary version appeared in Proc. 11th ACM Symp. on Principles of Distributed Computing, August 1992.

[Pel90] D. Peleg. Distributed data structures: A complexity-oriented structure. *4rd International Workshop on Distributed Algorithms*, 1990.

[Pet83] G.L Peterson. Concurrent reading while writing. *ACM Transactions on Programming Languages and Systems*, 5:46–55, 1983.

[Plo89] S. A. Plotkin. Sticky bits and universality of consensus. In *Proc. 8th ACM Symp. on Principles of Distributed Computing*, pages 159–175, August 1989.

[Sag85] Y. Sagiv. Concurrent operations on B-trees with overtaking. In *ACM Symp. on Principles of Database Systems*, pages 28–37, January 1985.

Acknowledgements

This research was supported in part by the fund for promotion of research at the technion. A preliminary version of this paper appeared in Proc. 13th ACM Symp. on Principles of Distributed Computing.