Information and Computation **153**, 117–138 (1999)
Article ID inco.1999.2793, available online at http://www.idealibrary.com on IDEAL®

The Power of Multiobjects[1]

Yehuda Afek[2]

Computer Science Department, Tel-Aviv University, Tel-Aviv 69978, Israel and AT&T Labs, 180 Park Avenue, Florham Park, New Jersey 07932

Michael Merritt[2]

AT&T Labs, 180 Park Avenue, Florham Park, New Jersey 07932

and

Gadi Taubenfeld

The Open University, 16 Klausner Street, P.O.B. 39328, Tel-Aviv 61392, Israel and AT&T Labs, 180 Park Avenue, Florham Park, New Jersey 07932

We consider shared memory systems that support multiobject operations in which processes may simultaneously access several objects in one atomic operation. We provide upper and lower bounds on the synchronization power (consensus number) of multiobject systems as a function of the type and the number of objects that may be simultaneously accessed in one atomic operation. These bounds imply that known classifications of component objects fail to characterize the synchronization power of their combination. In particular, we show that in the context of multiobjects, *fetch* & *add* objects are less powerful than *swap* objects, which in turn are less powerful than *queue* objects. This stands in contrast to the fact that *swap* can be implemented from *fetch* & *add*. Herein we introduce a restricted notion of implementation, called direct implementation. We show that, if objects of type *Y* have a direct implementation from objects of type *X*, then *Y*-based multiobjects can also be implemented from *X*-based multiobjects. Using this observation, we derive results such as: there are no *direct* implementations of *swap* or *queue* objects from any collection of commutative objects (e.g., *fetch* & *add*, *test* & *set*). © 1999 Academic Press

[1] A preliminary version of this work appeared in the "Proc. of the 15th ACM Symp. on Principles of Distributed Computing (PODC)," May 1996, pp. 213–223.

[2] Partial support provided by the Israel/United States Binational Science Foundation, Grant 94002.

1. INTRODUCTION

A shared memory system is a collection of objects accessed by a collection of processes. Generally, it is assumed that each process accesses the individual objects atomically and one at a time and that the individual accesses by different processes are interleaved. It is possible to consider alternatives to this serial access method, such as permitting processes to access multiple objects in a single atomic step and that these multiple accesses by different processes are interleaved. Each of these alternative views of shared memory can be understood as a different way of combining objects. This leads naturally to the idea of alternative object *combinators*, constructors that produce *compound* objects from sets of *component* objects.

This paper focuses on the properties of a specific parallel combinator, the *multiobject* combinator which, given a shared memory system with a set of component objects O and a parameter m, produces a compound object O^m in which processes are allowed to simultaneously (and atomically) execute operations on up to m of the component objects in O. An example is a *register* multiobject which allows processes to read or write up to m registers in a single atomic operation [MT94]. This object generalizes the *m-assignment* objects discussed by Herlihy [Her91] (which support writes to m registers in a single atomic operation) and *snapshot* objects [AAD$^+$93, And94, And93] (which support reads of multiple registers in a single atomic operation). In [Pat71], Patil has defined a strong type of semaphore, called a PV-multiple, which enables one to access several basic semaphores in one step.

We narrow our attention to the synchronization power (consensus number) of multiobjects, as a function of the types of the components and of m. (The consensus number of an object type is the largest number of processes which can solve the consensus problem using any number of objects of that type, plus atomic registers [Her91].) Although the consensus number of O^m may depend on the number of objects in O, this paper studies the impact of the *kinds* of objects in O (e.g., sets of *swap* versus sets of *queue* objects), not the *number* of such objects. Hence, we assume that each set contains sufficient copies of the component objects for the algorithms we present, and each impossibility result holds for an infinite number of component objects. For example, we show that if m different *swap* objects can be accessed in a single atomic step, then *n-consensus* (n processes consensus) can be implemented for as many as $n = \sqrt{2m+5}$ processes, but no such algorithm exists if $n > (2/\sqrt{3})\sqrt{2m+4} + \frac{1}{3}$. Previous research on the synchronization power of shared objects studied the dependence of such power on the type of the object [Her91], the total number of objects available [Jay93], and their size [AS94]. Our research studies another dimension: how the power varies with the number of objects that are accessed atomically in one step.

1.1. *Summary of Results*

Table 1 summarizes properties of some specific multiobjects. Given a set of objects O, O^m denotes the result of applying the multiobject combinator to the objects in O. In the table and throughout this paper, we also use the notation T^m,

TABLE 1

Bounds on Con(X^m), the Consensus Number of Multiobjets X^m, in Which up to m Component Objects of Type X Can Be Simultaneously Accessed

Multiobject X^m	Lower bound on Con(X^m): X^m implements consensus for at least this many processes	Upper bound on Con(X^m): No implementation of consensus from X^m exists for more than this many processes
Fetch & add^m [a]	2	2
Swap^m	$\lfloor\sqrt{2m+5}\rfloor$	$\frac{2}{\sqrt{3}}\sqrt{2m+4}+\frac{1}{3}$
c-consensus^m, $c>2$	$1+\left\lfloor\frac{c}{4}\left(\sqrt{1+4m-\frac{12m}{c}+\frac{8m}{c^2}}-1\right)\right\rfloor$	$1+c\sqrt{2m}$
Register^m [Her91]	$2m-2$	$2m-2$
Queue^2	∞	∞

[a] The same bound holds for any collection of commutative objects, such as *fetch & increments* and *test & set*.

where T is an object *type*, not a set of objects. Hence, by T^m we mean the class of objects that can be obtained by applying the multiobject combinator to sets of objects of type T.

Some general properties of the multiobject combinator emerge from this investigation. They imply, for example, that the consensus number of the component objects is in general inadequate to explain the properties of their multi-object combination (e.g., the consensus number of the combination). For example, the consensus number of *fetch & add* multiobjects is 2, the consensus number of *swap* multiobjects grows as the square root of the number of components that can be accessed simultaneously, and the consensus number of *queue* multiobjects is infinite, if as few as two of the queue objects can be accessed simultaneously. Yet, the three component objects each have consensus number 2. We show that similar examples occur throughout the consensus hierarchy—see the discussion in Section 6 for specifics. Hence, determining the consensus number of O^m affords a means of classifying objects that can be more discriminating than the consensus number of O alone.

Another theme that emerges from our investigation is the effect of supporting *read* operations. For many objects, the consensus number increases with the number of processes that can read the object. This is suggested in the table in the distinctions between *fetch & add*, *swap*, and *c-consensus*. Intuitively, *fetch & add*^{m} allows two processes to reach consensus, but no other process can read the result, while we show that *swap*^{2} allows two processes to reach consensus, and a third to read the result. Finally, when $c>2$, *c-consensus* objects can be used to allow $c-r$ processes to reach consensus, and r other processes to read the result for any r, $1\leqslant r<c$. As we discuss, the consensus number is also greatly affected by adding read operations directly to specific objects, e.g., *queues*. (Adding operations to objects is an example of another object combinator as discussed in Subsection 5.2.1.)

The standard notion of object implementation depends crucially on the fact that when one object type X implements another object type Y (denoted as $X\rightarrow Y$), then in any memory system in which instances of Y are combined in the standard way with objects of other types, each instance of Y can be replaced by sufficiently many instances of X, of *registers*, and by algorithms run by the processes in place of direct accesses to Y, resulting in a memory system that simulates the original. Such a substitution of X for Y when $X \rightarrow Y$ is not possible in multiobjects. That is, in general $X \rightarrow Y$ does not imply $X^m \rightarrow Y^m$. For example, it is known that *fetch & add* $\rightarrow$ *swap* [AWW93], but it follows from the upper bound for *fetch & add*^{m} and the lower bound for *swap*^{m} that *fetch & add*$^m \nrightarrow$ *swap*^{m}, for $m \geqslant 2$. Similarly it follows that *swap*$^m \nrightarrow$*queue*^{m} for $m \geqslant 2$.

However, it is possible to define a more constrained implementation relation, which does allow one object to replace another within multiobject constructions. We call this relation, defined in the next section, the *directly implements* relation, and we write it $X \xrightarrow{\text{di}} Y$. This relation satisfies two key composition properties; if $X \xrightarrow{\text{di}} Y$, then both $X^m \xrightarrow{\text{di}} Y^m$ and $X \rightarrow Y$, and, hence, also $X^m \rightarrow Y^m$. As a consequence, *fetch & add* $\overset{\text{di}}{\nrightarrow}$ *swap* (cannot directly implement) and *swap* $\overset{\text{di}}{\nrightarrow}$ *queue*. Interestingly these $\overset{\text{di}}{\nrightarrow}$ relations parallel the relative difficulties in constructing these objects from each other as reported in [AWW93]; i.e., the construction of *fetch & add* is relatively simpler than that of *swap*, and the authors of [AWW93] were not able to find a construction of *queue* from either *swap* or *fetch & add*.

We show how to use the $\xrightarrow{\text{di}}$ relation to obtain modular constructions of consensus algorithms using multiobjects. For example, we present modular constructions of *n-consensus* algorithms from *swap*^{m}, *c-consensus*^{m}, and *register*^{m} objects. These modular constructions hinge on the definition of (f, r)-*consensus* objects, which implement consensus for f processes and allow r other processes to read the result. We first present an *n-consensus* algorithm using (f, r)-*consensus*^{m} objects, where n is at most $(2-r)/2+\sqrt{(f-r/2)^2+(f-1)\,rm}$. Implementations of *n-consensus* from *swap*^{m}, *c-consensus*^{m}, and *register*^{m} objects are then derived by presenting direct implementations of $(2, 1)$-*consensus* from *swap*^{2}, $(c-r, r)$-*consensus* from *c-consensus*, and $(2, r)$-*consensus* from *register*^{2}.

Definitions of multiobjects and different implementation relations and their properties are provided in Section 2. In Section 3 we describe several *n-consensus* algorithms using different multiobjects and in Section 4 we present impossibility results to bound the algorithmic results of Section 3, and a general observation based on these results. Section 5 closes with a discussion of additional object combinators (such as adding or removing operations from an object). It is possible to construct many familiar objects from a few simple initial components, and simple object combinators. We suggest that a general theory of shared memory may emerge from a better understanding of such simple components and object combinators.

2. THE MULTIOBJECT COMBINATOR

The notion of one object implementing another is central to any theory of distributed or concurrent computation. One formulation of a standard notion of wait-free implementation [HW90] defines objects as I/O automata [LT87] and says that object type X implements object type Y if a wait-free algorithm exists which accesses instances of X and of shared registers, resulting in a simulation of an object of type Y. We write $X \rightarrow Y$ when such a relation exists between object types X and Y. We also write $\mathrm{Con}(X)$ to be the maximum n such that $X \rightarrow$ *n-consensus*, or ∞ if $X \rightarrow$ *n-consensus* for all n.

As we discuss in the introduction, it is important to consider whether $X \rightarrow Y$ implies $X^m \rightarrow Y^m$, and some of our results demonstrate this is not the case. For example, *fetch & add* $\rightarrow$ *swap* [AWW93, Wei94], but *fetch & add*$^m \not\rightarrow$ *swap*^{m} (because $\mathrm{Con}(\textit{fetch \& add}^m) = 2$ and $\mathrm{Con}(\textit{swap}^m) = O(\sqrt{m})$).

It is possible to define a more constrained implementation relation, *X directly implements Y*, or $X \xrightarrow{\text{di}} Y$, that does satisfy the property: $X \xrightarrow{\text{di}} Y$ implies $X^m \xrightarrow{\text{di}} Y^m$. In particular, X directly implements Y if X implements Y using a construction in which every operation op_y of Y is implemented by a series of operations on instances of X and of registers, and such that the linearization of the implementation of op_y can always be placed at the first access to an instance of X. (In the degenerate case in which Y can be implemented from registers alone, this definition can be applied by adding additional accesses to instances of X to designate appropriate linearization points within the implementations of the operations of Y.)

We state some simple properties satisfied by the direct implementation relation and the multiobject combinator. (See Section 5 for detailed definitions of objects and object combinators.)

THEOREM 1. *Let X and Y be two object types*:

1. $X \xrightarrow{\text{di}} Y$ *implies* $X \rightarrow Y$.
2. $X \xrightarrow{\text{di}} Y$ *implies* $X^m \xrightarrow{\text{di}} Y^m$ *for all* $m > 0$.
3. $X \xrightarrow{\text{di}} Y$ *implies* $X^m \rightarrow Y^m$ *for all* $m > 0$.
4. $X^m \xrightarrow{\text{di}} X^{m'}$ *for all* $m' > 0$.
5. $X^{pq} \xrightarrow{\text{di}} (X^p)^q$ *for all* $p, q > 0$.
6. $(X^p)^q \xrightarrow{\text{di}} X^{pq}$ *for all* $p, q > 0$.
7. $X \xrightarrow{\text{di}} Y$ *and* $Y \xrightarrow{\text{di}} Z$ *implies* $X \xrightarrow{\text{di}} Z$.
8. $X^p \xrightarrow{\text{di}} Y$ *and* $Y^q \rightarrow Z$ *implies* $X^{pq} \rightarrow Z$ *for all* $p, q > 0$.
9. $X^p \xrightarrow{\text{di}} Y$ *implies* $\mathrm{con}(Y^q)$ *for all* $p, q > 0$.
10. $\mathrm{Con}(X^m) < \mathrm{Con}(Y^m)$ *for any* $m > 0$ *implies* $X \overset{\text{di}}{\not\rightarrow} Y$.

Proof. All proofs are straightforward; we illustrate by outlining a proof of part 2, $X \xrightarrow{\text{di}} Y$ implies $X^m \xrightarrow{\text{di}} Y^m$ for all $m > 0$. The implementation of Y^m uses m sets of objects (of type X and registers), one for each component of Y. The implementation of a Y^m operation proceeds independently on each component Y operation up to the first instance of an operation on an object of type X. (For each component operation, this will involve some finite number of register operations.) When all m component operations are first prepared to access an object of type X, the accesses are all performed as a single X^m operation. After this, the component operations proceed independently again, with individual accesses to registers and objects of type X. Because $X \xrightarrow{\text{di}} Y$, each implemented Y operation can be linearized at the same point, the X^m operation. Hence $X^m \xrightarrow{\text{di}} Y^m$. ▮

3. IMPLEMENTATIONS OF *CONSENSUS* USING MULTIOBJECTS

We prove lower bounds on the consensus number of various multiobjects X^m by presenting consensus algorithms that access only X^m objects and read/write registers. Generally, the presentations assume a fixed number of processes, n, and describe a general algorithm using simultaneous access to sufficiently many instances of X.

The algorithms for implementing *n-consensus* from $swap^m$, $register^m$, and *c-consensus*^{m} objects are presented as reductions into one basic algorithm that implements *n-consensus* from a newly defined object, called (f, r)*-consensus*. The basic algorithm implements *n-consensus* using (f, r)*-consensus*^{m} multiobjects and is a generalization of Herlihy's implementation from *m-assignment* objects [Her91]. By part 8 of Theorem 1, the basic algorithm can be combined with direct implementations of (f, r)*-consensus* from $swap^2$ or $register^2$ to implement *n-consensus* from $swap^{2m}$ and $register^{2m}$. Despite technical difficulties, a similar construction can be used to replace $(c-r, r)$*-consensus* objects with *c-consensus* objects in the basic algorithm, implementing *n-consensus* from *c-consensus*^{m}.

3.1. *Implementation of n-consensus from* (f, r)*-consensus Objects*

Specifically, an (f, r)*-consensus* object is accessed by no more than $(f+r)$ distinct processes, f of which may invoke the consensus operations *propose*(0) or *propose*(1) and r of which may invoke *read* operations. A read linearized before any consensus operation returns the value $\perp$; otherwise, it returns the consensus value (0 or 1). (The consensus operations must be linearized as in an *f-consensus* object; i.e., the reads have no apparent effect on the object state.)

THEOREM 2. *For any f, m, $n \geqslant 2$ and $r \geqslant 1$, if $m \geqslant \lceil \lceil n/2 \rceil / \lfloor f/2 \rfloor \rceil \lceil (n-f)/r \rceil$, then* $\mathrm{Con}((f, r)$*-consensus*$^m) \geqslant n$.

Proof. Given a sufficient number of (f, r)*-consensus*^{m} objects, we describe an algorithm for reaching consensus for n processes. To reach consensus the n processes iteratively use a basic building block that solves consensus among two groups of processes, under the restriction that the members of each group all propose the same value. (The two groups may or may not propose different values.) The basic building block is used to run a series of competitions among the processes, starting with pairs of groups of one process each. Each pair of groups reaching consensus in one round competes together as a single group in the next round, until two final groups of at most $\lceil n/2 \rceil$ processes compete.

Consensus between two groups. Let the two groups of processes competing in the basic building block be $P = \{p_1, ..., p_{n_P}\}$ and $Q = \{q_1, ..., q_{n_Q}\}$, where all members of the same group propose the same value. Without loss of generality, assume $n_P \geqslant n_Q$. The basic idea is that a process tries to reach consensus on the value of its group by competing against all the members of the other group, simultaneously, in one atomic step. Furthermore, this simultaneous step accesses enough (f, r)-*consensus* objects so any other processes will be able to later observe the result of the competition.

Let v_P and v_Q be the values proposed by members of P and Q, respectively. Each of P and Q is partitioned into $\lceil n_P/\lfloor f/2\rfloor\rceil$ and $\lceil n_Q/\lceil f/2\rceil\rceil$ subsets of $\lfloor f/2\rfloor$ and $\lceil f/2\rceil$ processes each, respectively. Let $\mathbf{P}=\{P_1, ..., P_{\lceil n_P/\lfloor f/2\rfloor\rceil}\}$, and $\mathbf{Q}=\{Q_1, ..., Q_{\lceil n_Q/\lceil f/2\rceil\rceil}\}$ be the corresponding collections of subsets. Each member of P competes with all members of Q by accessing (f, r)-*consensus* objects. This can be done by arranging $\lceil n_P/\lfloor f/2\rfloor\rceil\lceil n_Q/\lceil f/2\rceil\rceil$ (f, r)-*consensus* objects in a two-dimensional array $COMPETE(1..\lceil n_P/\lfloor f/2\lfloor\rceil, 1..\lceil n_Q/\lceil f/2\rceil\rceil)$. Each process in P_i simultaneously proposes v_P to the $\lceil n_Q/\lceil f/2\rceil\rceil(f, r)$-*consensus* objects $COMPETE(i, j)$, for all j, $1\leqslant j\leqslant\lceil n_Q/\lceil f/2\rceil\rceil$, and similarly each process in Q_j simultaneously proposes v_Q to the $\lceil n_P/\lfloor f/2\rfloor\rceil$ (f, r)-*consensus* objects $COMPETE(i, j)$, for all i, $1\leqslant i\leqslant\lceil n_P/\lfloor f/2\rfloor\rceil$.

The outcome of each component (f, r)-*consensus* object $COMPETE(i, j)$ is apparent to the f members of $P_i\cup Q_j$, once they propose a value. The other n_P+n_Q-f processes in $(P\cup Q)-(P_i\cup Q_j)$ need to be able to read the outcome, but only r of them can access $COMPETE(i, j)$. Let $s=\lceil(n_P+n_Q-f)/r\rceil$. We need to replicate each (f, r)-*consensus* object in $COMPETE(i, j)$ a total of s times, and partition the n_P+n_Q-f process indices in $(P\cup Q)-(P_i\cup Q_j)$ into a collection of subsets $\mathbf{R}^{ij}=\{R_1^{ij}, ..., R_s^{ij}\}$, each containing at most r processes. Each subset is assigned one replica. Hence, the final data structure is the three-dimensional array $COMPETE(1..\lceil n_P/\lfloor f/2\rfloor\rceil, 1..\lceil n_Q/\lceil f/2\rceil\rceil, 1..s)$, illustrated in Fig. 1. Each process p in P_i simultaneously proposes v_P to the $s\lceil n_Q/\lceil f/2\rceil\rceil$ (f, r)-*consensus* objects $COMPETE(i, j, k)$ for all j, $1\leqslant j\leqslant\lceil n_Q/\lceil f/2\rceil\rceil$, and all k, $1\leqslant k\leqslant s$, and, similarly each process in Q_j simultaneously proposes v_Q to the $s\lceil n_P/\lfloor f/2\rfloor\rceil$ (f, r)-*consensus* objects $COMPETE(i, j, k)$ for all i, $1\leqslant i\leqslant\lceil n_P/\lfloor f/2\rfloor\rceil$, and all k, $1\leqslant k\leqslant s$. We call these steps the *write steps* for the building block in this round of the algorithm.

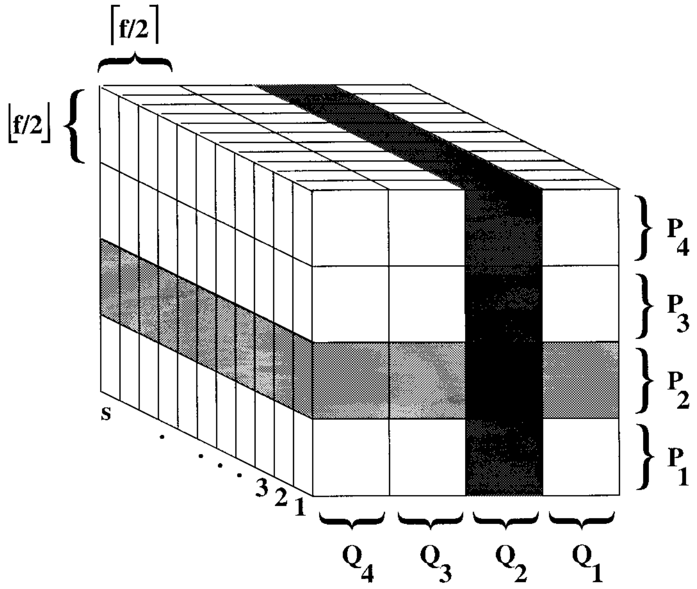

FIG. 1. Three-dimensional array of (f, r)-*consensus*^{m} objects in the basic *n-consensus* algorithm.

The first member of either group to take a write step determines the output of the competition for all processes in P and Q, by setting all copies of all (f, r)-*consensus* objects in a row or column of *COMPETE* to v_P or v_Q. Hence, if a process's write step returns the same value, v, from all component objects, it can return v as the outcome of this step of the algorithm.

If a process $p \in P_i \in \mathbf{P}$ returns different values from different component objects, it knows both that $v_P \neq v_Q$ and that some of the processes in Q have already taken a write step. Specifically, if any object $COMPETE(i, j, k)$ returns v_Q, process p reads the $\lceil n_P/\lfloor f/2 \rfloor \rceil - 1$ (f, r)-*consensus* objects $COMPETE(h, j, t_h)$ for $h = 1, ..., i-1, i+1, ..., \lceil n_P/\lfloor f/2 \rceil \rfloor$, and $p \in R^{h,j}_{t_h}$, one at a time. If all of these reads return v_Q, then p returns v_Q. Otherwise, p chooses another object $COMPETE(i, j', k')$ which returned v_Q during its write step and performs a similar series of reads, returning v_Q if they all return v_Q. Continuing in this way, if p exhausts all the *COMPETE* objects which returned v_Q during its write step, without returning v_Q, then p returns v_P.

Suppose that $COMPETE(i, j, k)$ returns v_Q during p's write step. Then p knows that at least one process in Q_j has taken a write step before p's, and hence is a candidate process to have taken a write step before all the processes in P. Let q be the first member of Q_j to take a write step. Next p has to determine whether q indeed took its write step before any member of P. Process q took its write step before all members of P if and only if all of the reads of the $\lceil n_P/\lfloor f/2 \rfloor \rceil - 1$ (f, r)-*consensus* objects $COMPETE(h, j, t_h)$, $h = 1, ..., i-1, i+1, ..., \lceil n_P/\lfloor f/2 \rfloor \rceil$, and $p \in R^{h,j}_{t_h}$, return v_Q. Hence, p can return v_Q from this step if this occurs. If one of these reads returns v_P, then some member of the associated subset of P performed its write step before q. Hence, if every such series of reads returns at least one v_P value, then no member of Q wrote before every member of P, and p can safely return v_P.

The full array contains $s \lceil n_P/\lfloor f/2 \rfloor \rceil \lceil n_Q/\lceil f/2 \rceil \rceil$ (f, r)-*consensus* objects, where $s = \lceil (n_P + n_Q - f)/r \rceil$. During a write step, processes need to simultaneously access as many as $s \lceil n_P/\lfloor f/2 \rfloor \rceil$ of these objects.

A complication arises if either n_P or n_Q are not divisible by $\lfloor f/2 \rfloor$ or $\lceil f/2 \rceil$, respectively. Suppose, for example, that there were one $p \in P$ and one $q \in Q$ left over after distributing the processes in P and Q in groups of $\lfloor f/2 \rfloor$ and $\lceil f/2 \rceil$, respectively. Then p and q contend alone in their (f, r)-*consensus* object, and there would be $n_P + n_Q - 2$ other processes needing to read the outcome, instead of $n_P + n_Q - f$. With only r readers per object, there would need to be $\lceil (n_P + n_Q - 2)/r \rceil$ copies of this object, instead of $s = \lceil (n_P + n_Q - f)/r \rceil$ copies. But observe that in the algorithm no consensus object is read by a process unless at least one other process has taken a write step that accesses it. This means that $f - 2$ processes can be assigned to use the consensus operation propose(v) to read this object, leaving $n_P + n_Q - f$ remaining readers.

The final round of the competition in which each group contains at most $\lceil n/2 \rceil$ processes requires the most (f, r)-*consensus* objects to be accessed simultaneously: the full array contains as many as $\lceil \lceil n/2 \rceil/\lfloor f/2 \rfloor \rceil \lceil \lfloor n/2 \rfloor/\lceil f/2 \rceil \rceil \lceil n - f/r \rceil$ (f, r)-*consensus* objects and, during their write step, processes need to simultaneously access as many as $\lceil \lceil n/2 \rceil/\lfloor f/2 \rfloor \rceil \lceil (n-f)/r \rceil$ of these objects, exactly as, assumed in the theorem. ∎

For fixed f, $m \geqslant 2$ and $r \geqslant 1$ the inequality of Theorem 2 is satisfied for any n such that $n \leqslant (2-r)/2 + \sqrt{(f-r/2)^2 + (f-1)\, rm}$. It follows that for fixed f, $m \geqslant 2$ and $r \geqslant 1$ there must be a consensus algorithm for $n \leqslant \lfloor (2-r)/2 + \sqrt{(f-r/2)^2 + (f-1)\, rm} \rfloor$ processes using (f, r)-*consensus*^{m} objects. This gives a lower bound on the consensus number of (f, r)-*consensus*^{m} objects.

COROLLARY 3. $\mathrm{Con}((f, r)\text{-}\mathit{consensus}^m) \geqslant \lfloor (2-r)/2 + \sqrt{(f-r/2)^2 + (f-1)\, rm} \rfloor$ *for any f, $m \geqslant 2$ and $r \geqslant 1$.*

3.2. Implementations of n-consensus from swap, register, and c-consensus objects

Implementations from swap and register. It is quite straightforward to directly implement $(2, 1)$-*consensus* from *swap*^{2}: Let p_0 and p_1 be the processes with access to the consensus operations *propose*(0) and *propose*(1), and let p_r be the reading process. The construction uses three swap objects, S_0, S_1, and S_{01}, initialized to $\perp$. Process p_0 proposes v_0 by swapping it into S_0 and S_{01} and process p_1 proposes v_1 by swapping it into S_1 and S_{01}. If either sees $\perp$ as the return value from S_{01}, it returns its input value. Otherwise, it returns the value returned from S_{01}. To read, process p_r swaps $\perp$ into S_0 and S_1. If both swaps return $\perp$, p_r knows the consensus value has yet to be determined. If exactly one swap returns a value other than $\perp$, or both return the same value, the reader knows that value is the outcome of the consensus. Finally, if two different values are returned, the reader swaps $\perp$ into S_{01}, and the read operation returns the value which was *not* returned by this final swap. (Since the consensus value is stable once determined, any subsequent *propose* or *read* operations do not need to access the shared memory.)

It is fairly obvious that this algorithm implements $(2, 1)$-*consensus*. Moreover, each operation can be linearized with its first *swap*^{2} operation. Hence, we have

THEOREM 4. $\mathit{swap}^2 \xrightarrow{\mathrm{di}} (2, 1)\text{-}\mathit{consensus}.$

COROLLARY 5. $\mathrm{Con}(\mathit{swap}^m) \geqslant \lfloor \sqrt{m + \frac{5}{4}} + \frac{1}{2} \rfloor$.

Proof. We focus on the case that m is odd and hence $\lfloor m/2 \rfloor = (m-1)/2$. (The analysis when m is even provides slightly better bounds on $\mathrm{Con}(\mathit{swap}^m)$.) By Theorems 1.4 and 1.9, $\mathrm{Con}(\mathit{swap}^m) \geqslant \mathrm{Con}(\mathit{swap}^{m-1})$. By Theorem 4, $\mathit{swap}^2 \xrightarrow{\mathrm{di}} (2, 1)$-*consensus*. Then Theorem 1.9 implies $\mathrm{Con}(\mathit{swap}^{m-1}) \geqslant \mathrm{Con}((2, 1)\text{-}\mathit{consensus}^{(m-1)/2})$, and by transitivity, $\mathrm{Con}(\mathit{swap}^m) \geqslant \mathrm{Con}((2, 1)\text{-}\mathit{consensus}^{(m-1)/2})$.

To bound the consensus number of $\mathrm{Con}((2, 1)\text{-}\mathit{consensus}^{(m-1)/2})$, we substitute 2, 1, and $(m-1)/2$ for f, r, and m, respectively, in the inequality of Theorem 2, getting the inequality $(m-1)/2 \geqslant \lceil n/2 \rceil (n-2)$. This inequality is satisfied by any $n \leqslant \lfloor \sqrt{m + \frac{5}{4}} + \frac{1}{2} \rfloor$. Hence, $\mathrm{Con}((2, 1)\text{-}\mathit{consensus}^{(m-1)/2}) \geqslant \lfloor \sqrt{m + \frac{5}{4}} + \frac{1}{2} \rfloor$. ∎

This result depends on component-by-component direct implementation of $(2, 1)$-*consensus* by *swap*^{2}. There is a more immediate and optimized implementation of the *COMPETE* array by *swap* multiobjects. (The direct implementation of $(2, 1)$-*consensus* by *swap*^{2} uses three swap objects, one of which determines which of two *swap*^{2} operations took place first, determining the consensus winner. The other two are used to determine whether both competing *swap*^{2} operations have taken place. In the context of the *COMPETE* array, these two swap objects and their associated operations can be shared, one for each row and column, instead of two per cell. This means $\lceil n/2+1 \rceil$ objects must be accessed simultaneously in each row or column, instead of n.) This optimized implementation of the *COMPETE* array results in a bound of $\mathrm{Con}(swap^m) \geqslant \sqrt{2m+5}$.

A lower bound of $m-2$ on the $\mathrm{Con}(register^m)$ follows from Theorem 2 using arguments similar to the above. However, stronger bound was obtained by Herlihy.

THEOREM 6 [Her91] $\mathrm{Con}(register^m) \geqslant 2m-2$.

Implementations from c-consensus. It is also possible, and seemingly quite direct, to simulate (f, r)-*consensus* objects with *c-consensus* objects. However, devising such a simulation is complicated by the fact that the *propose* operation permanently changes a *consensus* object, making it difficult to use in directly simulating a *read* operation. However, the (f, r)-*consensus*^{m} implementation in Theorem 2 satisfies the following property:

No component object is read until it has been accessed by a *propose* operation.

This leads naturally to the definition of a constrained environment, E, in which such an ordering of *propose* and *read* operations is guaranteed and in which we can require *c-consensus*^{m} objects to simulate (f, r)-*consensus*^{m} objects. Within this constrained environment, calls to (f, r)-*consensus*^{m} objects can be replaced directly with calls to *c-consensus*^{m} objects.

It is possible to condition the notions of $\rightarrow$ and $\xrightarrow{\mathrm{di}}$ on the environment E, obtaining weaker implementation relations $X \xrightarrow[E]{} Y$ and $X \xrightarrow[E]{\mathrm{di}} Y$ relating X and Y.

THEOREM 7. 1. *c-consensus*$^m \xrightarrow[E]{\mathrm{di}} (c-r, r)$-*consensus*^{m} *for any* r, $1 < c$, $1 \leqslant r < c-1$, *and any* $m > 0$.

2. *The implementation of n-consensus from* (f, r)-*consensus*^{m} *objects* (*from the proof of Theorem* 2) *satisfies E.*

3. *For all* $c > 2$, $m > 1$

$$\mathrm{Con}(c\text{-}consensus^m) \geqslant \max\{\mathrm{Con}((c-r, r)\text{-}consensus^m) : 1 \leqslant r \leqslant c-2\}$$
$$\geqslant \left\lfloor 1 + \frac{c}{4}\left(\sqrt{1 + 4m - \frac{12m}{c} + \frac{8m}{c^2}} - 1\right)\right\rfloor.$$

3.3. An Implementation of n-consensus from queue^2 Objects

The component *queue* object we consider supports the operations *enqueue* and *dequeue* by every process. A *dequeue* operation on an empty queue returns a special value, $\perp$. (The algorithm presented in this section never enqueues more than $2n+1$ values in any *queue* object, so by assuming the queues can hold at least this many values, the behavior of the component queue object when an enqueue is invoked on a full *queue* is immaterial.)

THEOREM 8. $\mathrm{Con}(queue^2) = \infty$.

Proof. We show that for any $n > 1$, *n-consensus* can be solved by n processes, $p_1, ..., p_n$, accessing a set of $4n$ queues, accessed two at a time. The algorithm is built out of n building blocks, $B_1, ..., B_n$, each B_i consisting of four queues, O_1^i, O_2^i,

```
**Implementation of building block $B_i$:**
Variables $r_1^k$, $r_2^k$, $r_3^k$, and $r_4^k$ are local variables of process $p_k$.
**Code for $p_i$ the owner of $B_i$:**
*enqueue* $i$ on $O_1^i$ and on $O_2^i$
*dequeue* from $O_2^i$ and $O_3^i$ into $r_2^i$ and $r_3^i$, respectively
**if** $r_3^i = \perp$ **then** return(*win*) /* No non-owner has enqueued. */
**else** return(*lose*)
**Code for $p_j$ a non-owner of $B_i$:**
*enqueue* $j$ on $O_1^i$ and on $O_2^i$
*enqueue* $(j, 0)$ on $O_3^i$ and on $O_4^i$
*enqueue* $(j, 1)$ on $O_3^i$ and on $O_4^i$
*dequeue* from $O_1^i$ and $O_2^i$ into $r_1^j$ and $r_2^j$, respectively
**if** $r_1^j = r_2^j$ **then** return(*win*) /* The owner hasn't dequeued. */
**else** /* The owner has dequeued.
            It remains to detect if before any enqueue by a non-owner*/
    {*dequeue* from $O_3^i$ and $O_4^i$ into $r_3^j$ and $r_4^j$, respectively
    **if** $r_3^j \neq r_4^j$ **then** return(*win*) /* The owner dequeued a queued value from $O_3^i$. */
    **else** return(*lose*) /* The owner dequeued $\perp$ from $O_3^i$. */
    }
```

FIG. 2. Implementation of building block B_i.

O_3^i, and O_4^i. Process p_i is the *owner* of building block B_i, and the remaining processes are *nonowners* of B_i. Owners and nonowners run different algorithms on each building block and return either *win* or *lose*. These algorithms guarantee that the nonowners always return a value different from the owner. Moreover, if the owner runs its algorithm before any nonowner, the owner returns *win*, and similarly, if a nonowner runs its algorithm before the owner, the nonowners return *win*.

We first describe the algorithms run by the owner and nonowner, and then describe how to use these building blocks to implement *n-consensus*.

The key to understanding the algorithm, given in Fig. 2, is that the owner's dequeue operation on O_3^i is racing the nonowner's first enqueue operations on O_3^i and O_4^i. Since each nonowner enqueues two values before dequeuing one, once an enqueue is made to O_3^i, this queue remains nonempty until the owner's dequeue. Hence, dequeuing $\perp$ or an enqueued value suffices to inform the owner whether his operation has run first. It remains for the non-owner to determine whether the owner or some nonowner has moved first.

After enqueuing its index into queues O_1^i and O_2^i, and twice queuing its index on queues O_3^i and O_4^i (using a bit to distinguish the two pairs of enqueued values), the nonowner first determines whether the owner has dequeued at all. The queues are always accessed in pairs $((O_1^i, O_2^i)$ and (O_3^i, O_4^i), respectively, except for the owner's dequeue of O_2^i and O_3^i.

Each process enqueues O_1^i and O_2^i and dequeus this pair at most once. Hence, each process will dequeue a (non-$\perp$) value from these queues. The owner's dequeue from only O_2^i misaligns these queues thereafter, ensuring that any pair of dequeues will return different values. Hence, a dequeue of these two objects suffices to tell whether the owner's dequeue has occurred. If not, the nonowner can safely return *win*. Hence, the nonowner only checks the O_3^i and O_4^i queues when it knows that the owner has indeed moved.

As observed above, note that the nonowners enqueue sufficiently many values before dequeuing to guarantee that no nonowner dequeues an empty queue. Moreover, the two queues, O_3^i and O_4^i, are always enqueued and dequeued in pairs by the nonowners, with distinct values enqueued each time. So the values dequeued from each are identical, unless the unpaired dequeue of O_3^i by the owner removed a value. Hence, as the comments indicate, a nonowner dequeuing distinct values from these queues is a clear indication that the owner dequeued a value (that was enqueued by some nonowner), misaligning the two queue contents for every later pair of dequeues, and dequeuing identical values must mean the owner dequeued an empty queue.

Constructing n-consensus from building blocks. Each process p_i first writes its input value to a shared register, $input_i$. Then, it runs its owner algorithm on building block B_i. Next, it runs its nonowner algorithms on the other building blocks. Since every process runs its owner algorithm before its nonowner algorithms, some owner returns *win*. Let j be the minimum process index of the blocks in which the owner *won*. Each process p_i determines j as follows: If p_i returned *win* in B_i and in every block with index less than i, then $j = i$. Otherwise, j is the minimum process index of a block in which p_i returned *lose*. Process p_i returns the value it reads in $input_j$. The agreement properties of the building blocks guarantee that all processes will choose the same value for j and, hence, of $input_j$. ∎

4. IMPOSSIBILITY PROOFS FOR MULTIOBJECTS

This section presents three impossibility proofs for multiobjects. Following Herlihy [Her91], we first define the class of *commutative* objects, which he showed have consensus number at most 2. We show that multiobjects constructed out of commutative components are themselves commutative and, hence, have no greater consensus number.

Next, we prove a general impossibility proof for objects built from ℓ-access-limited components (objects which can only be accessed by $\ell < n$ processes). This proof applies directly to (f, r)-*consensus* and c-*consensus* objects which are $f + r$ and c-access-limited, respectively. Finally, we close the section with an impossibility proof specific to the *swap*^{m} multiobject.

4.1. *Commutative Objects*

A shared memory object O is *commutative* if the relative order of any two operations cannot be determined by the nonparticipating processes. That is, let o_p and o_q be operations offered by O to processes p and q. Let $xo_p o_q y$ be a run in the sequential specification of O, where y contains no operations of p or q. The object O is commutative if for all pairs of operations o_p and o_q and runs $xo_p o_q y$, there exist operations o'_p and o'_q, such that $xo'_q o'_p y$ is also a run of O. Examples of commutative objects include test & set, fetch & add, fetch & complement, and all 2-access-limited objects, such as 2-*consensus*. The following is a simple property of multiobjects.

LEMMA 4.1. *Any multiobject constructed of commutative component objects is commutative.*

An immediate consequence of this lemma and Herlihy's observation [Her91] is

THEOREM 9. *If X is a commutative object then the consensus number of X^m is no greater than 2.*

4.2. *Impossibility Proof for ℓ-Access-Limited Objects*

Next, we prove a general impossibility proof for objects built from ℓ-access-limited components. Recall that these are objects which can only be accessed by $\ell < n$ processes.

THEOREM 10. *If O is a set of ℓ-access-limited objects, then* $\mathrm{Con}(O^m) < \ell\sqrt{2m} + 1$.

Proof. Note first that operations on 2-access-limited objects commute. From Theorem 9, for such objects, $\mathrm{Con}(O^m) \leqslant 2$. Assume then that $\ell > 2$. Suppose we have an algorithm for implementing consensus among n processes, using simultaneous access to up to m copies of ℓ-access-limited objects, where $\ell < n$.

Following [FLP85], we define a prefix of a run of a consensus algorithm (and the state in which it ends) to be *univalent* with value v if no process ever returns $\neg v$ in extensions of the run. If a prefix (and state) is not univalent, we say it is *bivalent*.

In any wait-free consensus algorithm, it is always possible to find a finite bivalent run x from which any step by any process moves to a univalent state. Then steps by processes from the set $P = \{p_1, ..., p_{n_0}\}$ make x 0-valent, and steps by processes from $Q = \{q_1, ..., q_{n_1}\}$ make x 1-valent, and $n = n_0 + n_1$. For process p_i or q_i, let O_{p_i} and O_{q_i} respectively denote the set of objects accessed in their next step after x. Without loss of generality, assume $n_1 \geqslant n/2$.

We show below that $|O_{p_1}| \geqslant n(n-2)/2(\ell-1)(\ell-2)$. Since $m \geqslant |O_{p_1}|$, we have $m \geqslant n(n-2)/2(\ell-1)(\ell-2)$. In turn, this implies $n \leqslant \sqrt{2m(\ell-1)(\ell-2)+1} + 1 < \ell\sqrt{2m} + 1$. The theorem follows.

It remains to prove the claim that $|O_{p_1}| \geqslant n(n-2)/2(\ell-1)(\ell-2)$. Using each processor name to denote its actions if run a single step, we have that xp_1q_i and xq_ip_1 are runs of opposite valencies which leave every object r in the same state unless r is in $O_{p_1} \cap O_{q_i}$. In each of these runs there are $n-2$ processes who have not yet taken steps after x, specifically the members of $\{p_2, ..., p_{n_0}, q_1, ..., q_{i+1}, ..., q_{n_1}\}$. Denote these processes by $S = \{s_1, ..., s_{n-2}\}$, and let s be an arbitrary process in S.

Consider what happens when s runs after sp_1q_i or xq_ip_1. The actions taken by s will be the same in each case unless s accesses some object in $(O_{p_1} \cap O_{q_i})$; since xp_1q_i or xq_ip_1 have different valencies, s must eventually access such an object. It follows that all $n-2$ members of S can access at least one object in $(O_{p_1} \cap O_{q_i})$. Since each object can be accessed by at most ℓ processes and p_1 and q_i both access each object in $(O_{p_1} \cap O_{q_i})$, at least $(n-2)/(\ell-2)$ objects are each accessed by both p_1 and q_i in a single step. That is, $|O_{p_1} \cap O_{q_i}| \geqslant (n-2)/(\ell-2)$.

There are at least $n/2$ members of Q and each accesses at least $(n-2)/(\ell-2)$ different objects in their next step after x, making a total of at least $n(n-2)/2(\ell-2)$ distinct accesses by members of Q to objects in O_{p_1}. Each object r_j in O_{p_1} is accessed by p_1 and at most $\ell-1$ members of Q, so there must be at least $n(n-2)/2(\ell-2)$ members of O_{p_1} to accommodate all $n(n-2)/2(\ell-2)$ distinct accesses by members of Q. That is, $|O_{p_1}| \geqslant n(n-2)/2(\ell-1)(\ell-2)$. The claim and, hence, the theorem follow. ∎

We notice that *c-consensus* objects are c-access-limited, and hence, the upper bound on Con(*c-consensus*^{m}) follows from Theorem 10.

COROLLARY 11. Con(*c-consensus*^{m}) $< 2\sqrt{2m}+1$.

With the proof of this corollary, we can now demonstrate that for all even consensus numbers, the consensus power of an object X does not predict the consensus power of X^m.

THEOREM 12. *Let c be any fixed even integer greater than* 1. *There exist objects X and Y and an integer m such that* $\mathrm{Con}(X)=\mathrm{Con}(Y)=c$, *but* $\mathrm{Con}(X^m)\neq\mathrm{Con}(Y^m)$.

Proof. Herlihy showed that $\mathrm{Con}(register^m)=2m-2$ [Her91]. This, together with Theorems 1.6 and 1.9, gives us that or integers m_1 and m_2 greater than, 1, $\mathrm{Con}((register^{m_1})^{m_2}) \geqslant \mathrm{Con}(register^{m_1m_2}) \geqslant 2m_1m_2-2$. By Theorem 1.5 $register^{m_1m_2} \xrightarrow{\mathrm{di}} (register^{m_1})^{m_2}$. By Theorem 1.9, $\mathrm{Con}(register^{m_1m_2}) \geqslant \mathrm{Con}((register^{m_1})^{m_2})$. All together we get that $\mathrm{Con}((register^{m_1})^{m_2})=2m_1m_2-2$.

Now let $X=$ *c-consensus* and $Y=register^{(c+2)/2}$. Clearly, $\mathrm{Con}(X)=\mathrm{Con}(Y)=c$, and by Theorem 7 and Corollary 11, $\mathrm{Con}(X^m)=\Theta(\sqrt{m})$, while by the arguments above, $\mathrm{Con}(Y^m)=\Theta(m)$. ∎

4.3. *Impossibility Proof for swap^m*

Any number of processes may invoke operations on *swap* objects. Hence, the result of the previous subsection, bounding the consensus value of ℓ-access-limited objects, cannot be usefully applied to *swap*. Nevertheless, a direct argument obtains a bound within a small constant factor of the $\lfloor\sqrt{m+\frac{5}{4}}+\frac{1}{2}\rfloor$ bound in Corollary 5.

THEOREM 13. $\mathrm{Con}(swap^m) \leqslant (2/\sqrt{3})\sqrt{2m+4}+\frac{1}{3}$.

Proof. As in the previous theorem, find a finite bivalent run x from which any step by any process moves to a univalent state. Then steps by processes from the set $P=\{p_1,\ldots,p_{n_0}\}$ make x 0-valent, and steps by processes from $Q=\{q_1,\ldots,q_{n_1}\}$ make x 1-valent, and $n=n_0+n_1$. For process p_i or q_i, let O_{p_i} and O_{q_i} respectively denote the set of objects accessed in their next step after x.

We prove below that $|O_{p_1}| \geqslant \min_{\lceil n/2\rceil \leqslant n_1<n} n+nn_1-n_1^2/2-5n_1/2-1$. Since $m \geqslant |O_{p_1}|$, we have $m \geqslant \min_{\lceil n/2\rceil \leqslant n_1<n} n+nn_1-n_1^2/2-5n_1/2-1$. In turn, this implies $n \leqslant (2/\sqrt{3})\sqrt{2(m+\frac{25}{24})}+\frac{1}{3} < (2/\sqrt{3})\sqrt{2m+4}+\frac{1}{3}$. The theorem follows.

It remains to prove the claim that $|O_{p_1}| \geqslant \min_{\lceil n/2\rceil \leqslant n_1<n} n+nn_1-n_1^2/2-5n_1/2-1$.

Using each processor name to denote its actions if run a single step, we have that $xp_1q_iq_{i-1}\cdots q_1$ and $xq_ip_1q_{i-1}\cdots q_1$ are runs of opposite valencies, which leave every object r in the same state unless r is in $O_{p_1}\cap O_{q_i}$ and not in $\bigcup_{j=1}^{i-1} O_{q_j}$. In each of these runs there are $n-(i+1)$ processes who have not yet taken steps after x, specifically the members of $\{p_2, ..., p_{n_0}, q_{i+1}, ..., q_{n_1}\}$. Denote these processes by $s_1, ..., s_{n-(i+1)}$.

Consider what happens when s_1 runs after $xp_1q_iq_{i-1}\cdots q_1$ or $xq_ip_1q_{i-1}\cdots q_1$. The actions taken by s_1 will be the same in each case until s_1 first accesses some object, r_1, in $(O_{p_1}\cap O_{q_i})-\bigcup_{j=1}^{i-1} O_{q_j}$; this swap in turn overwrites the value returned to s_1. Suppose we run s_1 until just after this swap and use $xp_1q_iq_{i-1}\cdots q_1S$ and $xq_ip_1q_{i-1}\cdots q_1s_1$ to denote these two runs. These runs are now indistinguishable by the processes $s_2, ..., s_{n-(i+1)}$ until each reads an object other than r_1 in $(O_{p_1}\cap O_{q_i})-\bigcup_{j=1}^{i-1} O_{q_j}$. By an obvious induction, each process s_{k+1} must eventually access a distinct object $r_{k+1}\notin\{r_1, ..., r_k\}$, but in $(O_{p_1}\cap O_{q_i})-\bigcup_{j=1}^{i-1} O_{q_j}$, in order to distinguish $xp_1q_iq_{i-1}\cdots q_1s_1\cdots s_k$ from $xq_ip_1q_{i-1}\cdots q_1s_1\cdots s_k$. (As above, use s_k to denote the steps taken by s_k through its first access of r_k.) Moreover, we have that $|O-p_1\cap\bigcup_{j=1}^{n_1} O_{q_j}|\geqslant\sum_{j=1}^{n_1} n-(j+1)=nn_1-n_1-\sum_{j=1}^{n_1} j=nn_1-n_1^2/2-3n_1/2$.

Consider the runs $xp_1q_{n_1}\cdots q_1$ and $xq_{n_1}\cdots q_1$. These runs are indistinguishable to p_2 until it first accesses an object r_2 in $O_{p_1}-\bigcup_{j=1}^{n_1} O_{q_j}$. As above, suppose we run p_2 until just after this swap, and use $xp_1q_{n_1}\cdots q_1p_2$ and $xq_{n_1}\cdots q_1p_2$ to denote these two runs. These runs are now indistinguishable to the processes $p_3, ..., p_{n_0}$ until each reads an object other than r_2 in $[O_{p_1}-\bigcup_{j=1}^{n_1} O_{q_j}]$. As with the simple induction above, each process p_k, $k>2$, must eventually access a distinct object $r_k\notin\{r_2, ..., r_{k-1}\}$, but in $[O_{p_1}-\bigcup_{j=1}^{n_1} O_{q_j}]$, in order to distinguish $xp_1q_{n_1}\cdots q_1p_2\cdots p_{k-1}$ and $xq_{n_1}\cdots q_1p_2\cdots p_{k-1}$. Thus, there must be at least n_0-1 objects $\{r_2, ..., r_{n_0}\}$ in $[O_{p_1}-\bigcup_{j=1}^{n_1} O_{q_j}]$.

It follows that $|O_{p_1}|\geqslant n_0+nn_1-n_1^2/2-3n_1/2-1$. Using $n_0+n_1=n$ and $\lceil n/2\rceil\leqslant n_1<n$, we have $|O_{p_1}|\geqslant\min_{\lceil n/2\rceil\leqslant n_1<n} n+nn_1-n_1^2/2-5n_1/2-1$. The claim and, hence, the theorem follow. ∎

5. OTHER COMBINATORS AND OBJECTS

This paper focuses on the multiobject combinator, as an alternative to the more standard serial object combinator. Consideration of this alternative naturally raises the issue of the space of possible object combinators and possible characterizations of that space. In this section, we briefly sketch a framework for formally characterizing general object combinators, including the multiobject and serial combinators, and raise some questions about general combinators. This formal framework characterizes combinators as functions of canonical *sequential specifications* of the component objects as automata.

5.1. Sequential Specifications via Automata

Herlihy and Wing describe a method for defining atomic shared memory objects via *sequential specifications*, sets of strings describing the object behavior when there is no interleaving of operations by different processes [HW90]. These sets of strings can be specified via (Mealy) state machines, in which the state transitions are labeled by operation invocations and responses. The following definition is taken from [AGMT92].

DEFINITION 1. A sequential specification of an object is a *quintuple* $\langle Q, S, I, R, \delta \rangle$, where

Q is a (finite or infinite) set of states.

$S \subseteq Q$ is a set of initial states,

$I = (Inv_1, ..., Inv_n)$ is an n-tuple of sets, where each Inv_i is a set of symbols denoting the operation invocations by process i. Let $Inv = \bigcup_i Inv_i$.

$R = (Res_1, ..., Res_n)$ is an n-tuple of sets, where each Res_i is a set of symbols denoting the operation responses for process i. Let $Res = \bigcup_i Res_i$.

Define the set of operations by process i on $\mathcal{O}$ to be $Op_i = Inv \times Res$, all the two-character strings of invocations and responses by i, and let $Op = \bigcup_i Op_i$. Then $\delta \subset Q \times Op \times Q$ is the transition relation.

As an example, a *swap* object over a data domain V and accessed by n processes can be specified by a state machine with state set V, and the transition relation $\{(v, swap_i(w)\ return_i(v), w) \mid 1 \leqslant i \leqslant n\}$. Similarly, n-process *register* objects over V, $register_V$, can be specified with the same state set and transition relation $\{(v, write_i(w)\ return_i, w) \mid 1 \leqslant i \leqslant n\}, \bigcup \{(v, read_i return_i(v), v) \mid 1 \leqslant i \leqslant n\}$.

5.2. *Examples of Object Combinators*

The multiobject combinator can now be defined as a function of the sequential specifications of the component objects. We call two such specifications $X = \langle Q_X, S_X, I_X, R_X, \delta_X \rangle$ and $Y = \langle Q_Y, S_Y, I_Y, R_Y, \delta_Y \rangle$ *compatible* if $I_X \cap I_Y = \emptyset$ and $R_X \cap R_Y = \emptyset$.

Formal definition of the multiobject combinator. Let $\{X_j \mid j \in J\}$ be a set of compatible shared objects with index set J, where $X_j = \langle Q_j, S_j, I_j, R_j, \delta_j \rangle$. Let I^m and R^m be the collections of subsets of $\bigcup_{j \in J} I_j$ and $\bigcup_{j \in J} R_j$, respectively, containing at most m operation invocations or responses, no more than one from each I_j or R_j, and let Op^m denote the set of operations IR, $I \in I^m$ and $R \in R^m$, such that I and R contain invocations and responses from the same subset of J. Then $\{X_j \mid j \in J\}^m = \langle Q^J, S^J, I^m, R^m, \delta^m \rangle$, where $\delta^m = \{(\bar{u}, IR, \bar{v}) \mid IR \in Op^m$ and $\forall j \in I$ and R contain invocations and responses of X_j, then $(u_j, I_j R_j, v_j) \in \delta_j$, and $u_j = v_j$, otherwise.

DEFINITIONS OF OTHER COMBINATORS. Other operators can now be defined in terms of sequential specifications. Let $X = \langle Q_X, S_X, I_X, R_X, \delta_X \rangle$ and $Y = \langle Q_Y, S_Y, I_Y, R_Y, \delta_Y \rangle$ be compatible shared object specifications:

1. Standard object composition. $XY = \langle Q_X \times Q_Y, S_X \times S_Y, I_X \cup I_Y, R_X \cup R_Y, \delta_{XY} \rangle$, where $\delta_{XY} = \{((u_X, u_Y), Op, (v_X, v_Y)) \mid ((u_X, Op, v_x) \in \delta_X$ and $u_Y = v_Y)$ or $((u_Y, Op, v_Y) \in \delta_Y$ and $u_X = v_X)\}$.

2. The sequential product. If $Q_X = Q_Y = Q$ and $S_X = S_Y = S$, then $X; Y = \langle Q, S, I_X \times I_Y, R_X \times R_Y, \delta_{X;Y} \rangle$, where $\delta_{X;Y} = \{(u, (inv_X, inv_Y)(res_X, res_Y), w) \mid (u, inv_X res_X, v) \in \delta_X$ and $(v, inv_Y res_Y, w) \in \delta_Y\}$. And

3. a union combinator, $+$. If $Q_X = Q_Y = Q$ and $S_X = S_Y = S$, then $X + Y = \langle Q, S, I_X \cup I_Y, R_X \cup R_Y, \delta_X \cup \delta_Y \rangle$.

For example, for a given state space V with initial state v_0, it is natural to define the simple *n-process read* and *write* objects R_V and W_V, where the transition relation for R_V is $\{(v, read_i return_i(v), v) \mid v \in V, 1 \leqslant i \leqslant n\}$, and the transition relation for W_V is $\{(v, write_i(w)\ return_i, w) \mid v, w \in V, 1 \leqslant i \leqslant n\}$. Then we can construct the n-process *register* object over V, $register_V$, from R_V and W_V: $register_V = R_V + W_V$. Similarly, the *swap* object over V, $swap_V$, can be constructed from read and write objects via the sequential combinator: $R_V;\ W_V = swap_V$.

5.2.1. *The + combinator.* The + combinator is a very simple but powerful object combinator. To illustrate some of its properties, we briefly discuss some specific objects constructed with this combinator. For example, applying the + combinator to *swap* (over integers) and *fetch & add* results in an object, *swap + fetch & add*, which supports both the *swap* and *fetch & add* operations on the same memory location.

The consensus number of *enqueue + red* is ∞. A process first enqueues its input value and then reads the queue contents and decides on the value at the head of the queue. This solves consensus for any number of processes.

An obvious lower bound of $2m-2$ on the consensus number of $(swap + read)^m$ and of $(swap + fetch\ \&\ add)$ follows from Herlihy's consensus algorithm from *m-assignment*, which shows that $\mathrm{Con}(register^m) \geqslant 2m-2$ [Her91, MT94]. Since a *swap* operation is more powerful than a *write* operation, there is a direct implementation of *register* from *swap + read*, $swap + read \xrightarrow{\mathrm{di}} register$. Hence, $\{swap + read\}^m \rightarrow register^m$; we can simply replace any *write* operation by the corresponding *swap* operation in Herlihy's algorithm and get a consensus algorithm for $2m-2$ processes using $\{swap + read\}^m$ objects. A similar argument works for $\{swap + fetch\ \&\ add\}^m$ objects.

The above discussion gives a lower bound on $\mathrm{Con}(swap + read)^m$; however, the question of finding a bound b such that there is no consensus algorithm for more than b processes from $(swap + read)^m$ is left open. We note that the result reported in Section 3.2 ($\mathrm{Con}(swap^m) \geqslant \sqrt{2m+5}$) shows that $\mathrm{Con}(\{swap + read\}^2) \geqslant 3$. Hence, $\{swap + read\}^2$ is strictly more powerful than $register^2$.

5.3. The Multiobjects $safe^m$ and $regular^m$

Hitherto, we have considered the multiobject combinator as applied to atomic objects. This leaves open the question of how to extend the definition to objects which do not have linearizable specifications, such as safe or regular registers [Lam86]. Saying that a multisafe object permits atomic access to m safe registers does not make much sense, since safe registers are not themselves atomic objects. (And in particular, $safe^1$ would be an atomic, not a safe, register.)

5.3.1. *Safe registers.* In the case of a single safe register, when two processes are trying to write at the same the end result is that one of the values is eventually written. A read event that happens after these two writes terminate should return the current register value, while a read that is concurrent with a write may return any value.

Now, what should happen in the case of a multisafe register, when two processes are trying to write concurrently to the same set of registers, say registers $r_1, ..., r_m$? The weakest sensible requirement seems to be that, once the write operations terminate, the value of each register r_i $(1 < i < m)$ is one of the two values that the processes tried to write into it. A stronger requirement would be that the values of all the m registers are written by the same process.

Thus, for example, if the first process tries to write "1" to all the registers and the other process tries to write "2," with the weaker requirement the value of each register after the writes is either "1" or "2," while with the stronger requirement either the values of *all* the registers are "1" or all are "2."

A simple observation is that it is possible to implement multiobjects that satisfy the weaker requirement by simply accessing the m safe registers sequentially. Thus, since the consensus number of safe registers is one, the consensus number of these trivial multiobjects is also one.

Since the weaker requirement is trivial, we choose to define multisafe registers as satisfying the stronger requirement and denote them by $safe^m$, where m is the maximum number of registers that a process can access in one step. Interestingly, we show that the consensus number of a $safe^m$ register is 1 (i.e., $\text{Con}(\text{safe}^m) = 1$ for any $m \geqslant 1$).

5.3.2. *Regular registers.* A regular register has a stronger specification than a safe register: in the single-reader/single writer case, a read operation that is concurrent with a write must return either the new or the old value; but successive read operations overlapping a single write operation may return any sequence of new and old values.

We define a multiregular register, $regular^m$, by first considering a specific implementation of a multireader/multiwriter regular register, *regular*. Each read and write operation of *regular* consists of a sequence of three atomic actions corresponding to the operation invocation, an internal operation, and the operation response. In the case of a write operation, the internal action updates an internal state variable with the value being written. In the case of a read operation, the internal action nondeterministically chooses the stored value *or* the value of any pending write operation (operations are pending if they have an invocation with no matching response). The value chosen by this internal step is later returned to the calling process.

An automaton specification of $regular^m$ is derived from *regular* in the natural way, in which each multioperation of $regular^m$ consists of a sequence of three atomic actions, each of which in turn are m-vectors of invocations, internal operations, and responses of the m *regular* component objects.

We show below that the consensus number of a multiregular register $regular^m$ is 1. Since multiregular registers directly implement multisafe registers, Theorem 1.9 implies the same bound for multisafe registers.

THEOREM 14. $\text{Con}(safe^m) = \text{Con}(regular^m) = 1$.

Proof. Using standard bivalency arguments [FLP85], we first show that there is no 2-process consensus algorithm using $regular^m$ objects. Assume there is a consensus algorithm for two processes p and q using *regular*^{m} objects. The standard FLP argument shows a bivalent configuration x must be reachable from which any multiatomic action (internal, invocation, or response) by either process reaches a univalent state. A case analysis on the possible pairs of internal actions concludes the proof.

First consider the case in which one of the multiatomic actions, say q_1 by process q is an invocation or response. There are runs by p from xq_1 that are indistinguishable from the run by p from x, thus reaching the same decision value, a contradiction. Notice that, although the atomic invocation or response by q may increase or decrease, respectively, the set of possible values read by an internal multiread operation by p, in either case there is a value which is readable both before and after either the invocation or the response.

Next consider whether two internal steps by the same process can take x to states of different decision values. (This case arises because the internal steps of a regular read operation choose the value read nondeterministically.) Specifically, suppose there are two internal atomic steps by q, q_1 and q_2 (where each of q_1 and q_2 consists of the internal read or write operations of m different component objects), such that xq_1 and xq_2 (the configurations reached from x after q_1 and q_2, respectively) have different decision values. At least one of the component read operations of q_1 and q_2 read different values (otherwise $q_1 = q_2$), but the write operations write identical values. It follows that any run by p (in particular any deciding run) from configuration xq_1 is indistinguishable by p from the corresponding run from xq_2, a contradiction. So xq_1 and xq_2 have the same deciding values, and by symmetry, any pair of possible next steps by p also have the same deciding values.

The final case to consider is that two internal steps by different processes take x to states of different deciding values. Indeed, since x is bivalent by assumption and, by the preceding paragraph any steps by a single process have the same decision value, it follows that for any single atomic steps p' and q' of p and q, respectively, xp' and xq' have opposite decision values. Let q_1 be a specific action of q that is possible from x. The read components of action q_1 are invisible to p. Moreover, any value that can be read after an internal write action of a *regular* object could also have been read before that action (as the value of some pending write). Hence, if $xq_1 p_1 \cdots p_r$ is a deciding run by p, then $xp_1 \cdots p_r$ is also a run and is indistinguishable by p from $xqp_1 \cdots p_r$. So $xp_1 \cdots p_r$ must be a deciding run with the same decision value as $xqp_1 \cdots p_r$ contradicting the argument above that xp_1 and xq_1 must have opposite decision values.

This concludes the proof that $\mathrm{Con}(\textit{regular}^m) = 1$. As observed above, since multiregular registers directly implement multisafe registers, Theorem 1.9 implies the same bound for multisafe registers. ▮

This observation has an interesting consequence, given that (atomic) registers can be implemented from regular registers.

Corollary 15. *For all $m \geqslant 1$, regular$^m \overset{\text{di}}{\not\rightarrow}$ register.*

6. CONCLUSIONS AND DISCUSSION

Sections 3 and 4 provide the results needed to demonstrate the phenomena mentioned in the Introduction: Theorem 9 implies Con(*fetch & add*^{m}) $= 2$, Corollary 5 and Theorem 13 imply Con(*swap*^{m}) $= \Theta(\sqrt{m})$, and by Theorem 8, Con(*queue*^{m}) $= \infty$. Hence, the consensus number of the original object (which for each of *fetch & add*, *swap*, and *queue* is 2), does not determine the consensus number of the corresponding multiobject.

Indeed, Theorem 12 demonstrates that this phenomenon occurs throughout the consensus hierarchy for any consensus number divisible by 2 and not only with objects having consensus number 2.

Following our work, Jayanti and Khanna have investigated the consequences of allowing an unbounded number of atomic operations within a single step—what we could denote X^∞. They point out that under this multiobject combinator, this divergence phenomenon does not apply to objects which directly implement 3-*consensus*: When arbitrarily many simultaneous atomic operations are allowed on objects which directly implement 3-consensus, the resulting multiobject has unbounded consensus number [JK97]. That is, by Theorems 1.2 and 7 for any object X such that $X \xrightarrow{\text{di}}$ 3-*consensus*, $\text{Con}(X^\infty) = \infty$. Obviously, if $X \xrightarrow{\text{di}}$ k-*consensus* for $k > 3$, then $X \xrightarrow{\text{di}}$ 3-*consensus*.

It is interesting to note that the number of processes that are able to read a component object X can have a larger effect on the consensus number of X^m than does the consensus number of X. For example, *register* and *register*^{m} have consensus numbers 1 and $2m - 2$, respectively, while both *fetch & add* and *fetch & add*^{m} have consensus number 2. In recent work, Ruppert has shown that if X is an object with consensus number $c > 2$, then $\text{Cons}(X^m) = \Omega(c\sqrt{m})$, and if X supports a read operation, then $\text{Cons}(X^m) = \Omega(cm)$ [Rup98].

There are several interesting questions about multiobjects that have not yet been explored. For example, what are their properties under other failure types and under failure models weaker than wait freedom? What is the impact of restricting the total number of multi-objects of a given type or the number of component objects? Does a characterization of specific simple objects and combinators, in a topological (or other) general setting, lead naturally to a characterization of their various combinations?

Although we have focused mainly on the multiobject combinator, there are other interesting combinators which could be investigated in order to better understand the nature of shared memory objects.

Finally, the ability to access multiple memory locations in one atomic step is useful in implementing concurrent lock-free data structures. For that reason, others have proposed lock-free implementations of atomic multiword operations [AD96, AMTT97, AM95, HM93, IR94, ST95]. Some of this work assumes the existence of special hardware [HM93], while others [AD96, AMTT97, AM95, IR94, ST95] assume the existence of a strong (universal) single word operation. Our work demonstrates that some multiobjects necessarily imply the availability of such universal objects. For example, it follows from our results that implementation of an object which supports atomic operations on two queues implies the existence of a universal single word operation, while atomic access to several test-and-set objects does not.

ACKNOWLEDGMENTS

We thank Faith Fich—the editor, Eric Ruppert, and the anonymous referees, who discovered several errors in earlier versions of this paper and made many useful suggestions. In particular we thank Faith for helping us to improve the paper through many iterations. The algorithm in the proof of Theorem 8 was simplified, based on one referee's suggestion.

Received August 16, 1996; final manuscript received March 1, 1999

REFERENCES

[AAD$^+$93] Afek, Y., Attiya, H., Dolev, D. Gafni, E., Merritt, M., and Shavit, N. (1993), Atomic snapshots of shared memory, *J. Assoc. Comput. Mach.* **40**(4), 873–890.

[AD96] Attiya, H., and Dagan, E. (1996), Universal operations: Unary versus binary, *in* "Proc. of the 15th ACM Symp. on Principals of Distributed Computing," pp. 223–232.

[AGMT92] Afek, Y., Greenberg, D., Merritt, M., and Taubenfeld, G. (1995), Computing with faulty shared memory, *Journal of the ACM* **42**(6), 1231–1274. [Also in *Proc. 11th Annual ACM Symp. on Principles of Distributed Computing*, pages 47–58, August 1992].

[AM95] Anderson, J. H., and Moir, M. (1995), Universal Constructions for Multi-Object Operations, *in* "Proc. of the 14th ACM Symp. on Principals of Distributed Computing," pp. 184–193.

[AMTT97] Afek, Y. Meritt, M. Taubenfeld, G., and Touitou, D. (1997), Disentangling multi-object operations, *in* "Proc. 1997 ACM Symp. on Distributed Computing," pp. 111–120.

[And93] Anderson, J. (1993), Composite registers, *Distrib. Comput.* **6**.

[And94] Anderson, J. (1994), Multi-writer composite registers, *Distrib. Comput.* **7**.

[AS94] Afek, Y., and Stupp, G. (1994), Delimiting the power of bounded size synchronization objects, *in* "Proc. 13th ACM Symp. on Principles of Distributed Computing," pp. 42–51.

[AWW93] Afek, Y. Weisberger, E., and Weisman, H. (1993), A completeness theorem for a class of synchronization objects, *in* "Proc. 12th ACM Symp. on Principles of Distributed Computing," pp. 159–170.

[FLP85] Fischer, M., Lynch, N., and Paterson, M. (1985), Impossibility of distributed consensus with one faulty process, *Journal of the ACM* **32**, 374–382.

[Her91] Herlihy, M. (1991), Wait-free synchronization, *ACM Trans. Programm. Lang. Systems* **13**(1), 124–149.

[HM93] Herlihy, M., and Moss, J. E. B. (1993), Transactional Memory: Architectural Support for Lock-Free Data Structures, *in* "20th Annual Symp. on Computer Architecture," pp. 289–300.

[HW90] Herlihy, M., and Wing, J. M. (1990), Linearizability: A correctness condition for concurrent objects, *ACM Trans. Programm. Lang. Systems* **12**(3), 463–492.

[IR94] Israeli, A., and Rappoport, L. (1994), Disjoint-Access-Parallel Implementations of Strong Shared Memory, *in* "Proc. of the 12th ACM Symp. on Principles of Distributed Computing," pp. 151–160.

[Jay93] Jayanti, P. (1993), On the robustness of Herlihy's hierarchy, *in* "Proc. 12th Symp. on Principles of Distributed Computing," pp. 145–158.

[JK97] Jayanti, P., and Khanna, S. (1997), On the power of multi-objects, *in* "Proc. 11th Int. Workshop on Distributed Algorithms, WDAG'97," LNCS 1320, pp. 320–332, Springer-Verlag, New York/Berlin.

[Lam86] Lamport, L. (1986), On interprocess communication, parts I and II, *Distrib. Comput.* **1**, 77–101.

[LT87] Lynch, N. A., and Tuttle, M. (1987), Hierarchical correctness proofs for distributed algorithms, *in* "Proc. of 6th ACM Symp. on Principles of Distributed Computation," pp. 137–151. [Expandend version available as Technical Report MIT/LCS/TR-387, April 1987]

[MT94] Merritt, M., and Taubenfeld, G. (1994), Atomic *m*-register operations, *Distrib. Computing* **7**, 213–221. [Also in "Proc. 5th Int. Workshop on Distributed Algorithms, 1991," pages 289-294]

[Pat71] Patil, S. S. (1971), Limitations of capabilities of Dijkstra's semaphore primitives for coordination among processes, *in* "Project MAC Computational Structures Group," MIT memo 57.

[Rup98] Ruppert, E. Consensus numbers of multi-objects, *in* "Proc. of the 17th Annual ACM Symp. on Principals of Distributed Computing, July 1998."

[ST95] Shavit, N., and Touitou, D. Software transactional memory, *in* "Proc. of the 14th Annual ACM Symp. on Principals of Distributed Computing, August 1995," pp. 204–213.

[Wei94] Weisman, H. (1994), "Implementing Shared Memory Overwriting Objects," Master's thesis, Tel-Aviv University.

Printed in Belgium