M. Li, J. Tromp, P.M.B. Vitányi

How to share concurrent wait-free variables

Computer Science/Department of Algorithmics & Architecture

Report CS-R8916 April

1989

M. Li, J. Tromp, P.M.B. Vitányi

How to share concurrent wait-free variables

Computer Science/Department of Algorithmics & Architecture Report CS-R8916 April

How to Share Concurrent Wait-Free Variables

Ming Li[*]
Computer Science Department, York University
North York, Ontario M3J 1P3, Canada

John Tromp
Paul M.B. Vitányi
Centrum voor Wiskunde en Informatica
Kruislaan 413, 1098 SJ Amsterdam, The Netherlands
and
Faculteit Wiskunde en Informatica
Universiteit van Amsterdam

April 19, 1989

Abstract

We present a solution to the problem of sharing data between multiple asynchronous users—each of which can both read and write the data—such that the accesses are serialisable and free from waiting. This allows a maximum of parallellism in distributed systems. By using a structured, top-down approach, we obtain a better understanding of what the algorithms do, why they do it, and that they correctly implement the specification. Our main construction of an n-user atomic variable directly from single-writer, single-reader atomic variables requires $O(n)$ control bits per subvariable and $O(n)$ accesses to subvariables per read/write action.

1980 Mathematics Subject Classification: 68C05, 68C25, 68A05, 68B20.
CR Categories: B.3.2, B.4.3, D.4.1, D.4.4.
Keywords and Phrases: Shared variable (register), concurrent reading and writing, atomicity, muliwriter variable, simulation.
Note: This paper is submitted for publication elsewhere.

[*]M. Li was supported in part by the National Science Foundation under Grant DCR-8606366 at Ohio State University, by Office of Naval Research Grant N00014-85-K-0445 and Army Research Office Grant DAAL03-86-K-0171 at Harvard University, and by NSERC Grant OGP-0036747 at York. The basic algorithm presented here was first announced in M. Li, P.M.B. Vitanyi, Tech. Report. TR 01-87, Aiken Comp. Lab., Harvard University, November 1987. E-mail adresses: li@yuyetti.bitnet, tromp@cwi.nl and paulv@cwi.nl

Report CS-R8916
Centre for Mathematics and Computer Science
P.O. Box 4079, 1009 AB Amsterdam, The Netherlands

1 Introduction

In [8] Lamport has shown how an atomic variable—one whose accesses appear to be indivisible—shared between one writer and one reader, acting asynchronously and without waiting, can be constructed from lower level hardware rather than just assuming its existence. Naturally, these ideas have aroused interest in the construction of multi-user atomic variables of that type. In a short time this has already led to a large number of conceptually extremely complicated ad hoc constructs and (erroneous) proofs. In this paper we will supply a uniform solution to all subproblems, given Lamport's construction, and derive the implementations by correctness-preserving transformations from the specification.

1.1 Informal Problem Statement and Main Result

Usually, with asynchronous readers and writers, atomicity of operations is simply assumed or enforced by synchronization primitives like semaphores. However, active serialization of asynchronous concurrent actions always implies waiting by one action for another. In contrast, our aim is to realize the maximum amount of parallellism inherent in concurrent actions by avoiding waiting altogether in our algorithms. In such a setting, serializability is *not* actively enforced, rather it is the result of a pre-established harmony in the way the executions of the algorithm by the various processors interact. Any one of the references, say [8] or [16], describes the problem area in some detail.

Our point of departure is the solution [12, 8] of the following problem. (We keep the discussion informal). A flip-flop is a Boolean variable that can be read (tested) by one processor and written (set, reset, or changed) by another. Suppose, we are given atomic flip-flops as building blocks, and are asked to implement an atomic variable with range 0 to $n-1$, that can be written by one processor and read by another one. Of course, $\lceil \log n \rceil$ flip-flops suffice to hold such a value. We stipulate that the two processors are asynchronous and do not wait for one another. Suppose the writer gets stuck after it has set half the bits of the new value. If the reader executes a read while the writer is stuck, it obtains a value that consists of half the new value and half the old one. Obviously, this violates atomicity.

We will use the following naming convention for classifying the accessibility of a variable (also called register):

single-user just a local variable

single-reader implicitly a single writer

multi-reader again a single writer

multi-writer implicitly multiple readers

multi-user each user can both write and read

This constitutes a hierarchy of variables[1]. At the outset we state our main result:

Theorem 1 *We supply a construction for an atomic n-user variable from $O(n^2)$ atomic 1-reader 1-writer variables, using $O(n)$ accesses of subvariables per operation execution and $O(n)$ control bits per subvariable.*

By operation we mean a higher level read or write action on the shared variable. The algorithm is developed from formal specifications in a structured programming style, such that correctness proof and algorithm development go hand in hand.

1.2 Comparison with Related Work.

Related ad hoc and very difficult constructions are given by [14, 7, 3, 10, 6] for the single-reader to multi-reader case (which we deal with in appendix B), and by [16, 11, 6, 4] for the multi-reader to multi-writer case (see appendix A). We note that especially the latter construction has appeared to be quite difficult. Both algorithms that have been completely published, and subjected to scrunity, turned out to contain errors. I.e., the algorithm in [16] presented in FOCS86 is not fully atomic but only satisfies the weaker "regularity" condition, as pointed out in FOCS87 errata. A modification of this algorithm presented subsequently in FOCS87 [11], was found to contain several errors by Russel Schaffer [13]. The multiwriter algorithm promised in [6] has not yet been published in any detail. The recent [4] starts from multi-reader variables, and uses the simplified version of the unbounded tag algorithm of [16] (presented here) as point of departure. Generally, papers in the area are hard to comprehend and check.

With these difficulties, there has been no previous attempt to implement an n-user variable directly from single reader variables, like we present here. Yet we believe the construction we present is relatively simple and transparent. Both problems above, that have been the subject of other investigations, are solved by simplifications (as it were "projections") of our main solution. The precise form of our solution was arrived at by implementing and empirically testing several candidates. Appendix C decribes the results of the *simulation* of our main algorithms. We appear to improve all existing algorithms under some natural measures of complexity. See table 1 for a complexity-wise comparison of our direct solution here with the best combinations.

Explanation: The compound variable (here n-user) is composed from primitive variables (here single-reader). To store a value in the compound variable, it is stored in a subset of the primitive variables, along with some control information (like timestamps) to allow selection of the most recent value. The

[1] Although a multi-user variable can be trivially implemented by a multi-writer variable (and is not more powerful in that sense), this requires in the worst case a quadrupling in the number of subvariables ("space") used. It also seems impractical to make a distinction between users restricted to write actions and users restricted to read actions.

paper	control bits	atomic accesses
[This paper]	$O(n^3)$	$O(n)$
[3] + [11]	$\Omega(n^3)$	$\Omega(n^3)$
[14, 7, 3, 6, 16]	$\Omega(n^3)$	$\Omega(n^2)$
[14, 7, 3, 4]	$\Omega(n^3)$	$\Omega(n^2 \log n)$

Table 1: worst case complexity comparison

'control bits' column displays the overall number of bits of control information required, summed over all primitive variables ("space complexity"). The 'atomic accesses' column displays the number of reads or writes on primitive variables required in the execution of one read or write on the compound variable ("time complexity"). The related work is [1, 2, 3, 4, 6, 7, 8, 9, 10, 11, 12, 14, 16, 15]

1.3 Definitions, the main Problem, and Specification

A concurrent system consists of a collection of sequential processes that communicate through shared data structures. The most basic such data structure is a shared variable. A user of such a variable V can start an action a (read or write) at any time when it is not engaged in another action, by invoking an "execute a" command on V, which finishes at some later time, possibly returning the value read. We can express the semantics in terms of a local value v of a process P and the global value contained in V. In absence of any other concurrent action the result of process P writing its local value v to V is that $V{:=}v$ is executed, and the result of a process reading the global V is that $v{:=}V$ is executed.

An *implementation* of V consists of a set of *protocols*, one for each reader and writer process, and a set of shared variables $X, Y, \ldots, Z$ (these are sometimes called *registers*). An operation execution a by user process P on V consists of an execution of the associated protocol which

- starts at time $s(a)$ (the start time)
- applies some transformations on the variables $X, Y, \ldots, Z$
- returns a result to P at time $f(a) > s(a)$ (the finish time)

We assume that start and finish times of different actions are all disjoint, i.e., for actions $a, b, a \neq b$ we have $s(a) \neq s(b)$, $s(a) \neq f(b)$ and $f(a) \neq f(b)$.

All interactions between processes and variables are asynchronous but reliable, and can be thought of as being mediated by a concurrent scheduler.

The read/write operations are total, i.e., they are defined for every state of the variable. An implementation is *wait-free* if the number of subvariable accesses in an operation execution is bounded by a constant, which depends only on the number of readers and writers.

Linearizability or atomicity is defined in terms of equivalence with a sequential system in which actions are mediated by a sequential scheduler that permits only one operation to execute on any variable at a time. A shared variable is *atomic*, if each read and write of it actually happens, or appears to take effect, instantaneously at some point between its invocation and response, irrespective of its actual duration. This can be formalized as follows.

Let V be a shared variable with associated user processes P,Q,...,R which execute a set of operation executions $\mathcal{A}$ on V. Order the set $\mathcal{A}$ (of reads and writes) such that action a precedes b, $a \rightarrow b$, if $f(a) < s(b)$. Note that with this definition, $\rightarrow$ is a special type of partial order called an *interval* order (that is, a transitive binary relation such that if $a \rightarrow b$ and $c \rightarrow d$ then $a \rightarrow d$ or $c \rightarrow b$). Define the reading mapping π as a mapping from reads to writes by: if r is a read that returns the value written by write w, then $\pi(r) = w$. We call the triple $\sigma = (\mathcal{A}, \rightarrow, \pi)$ a *system execution.*

Definition. A system execution σ is *atomic* if we can extend $\rightarrow$ to a total order $\rightarrow'$ such that

(A1) $\pi(r) \rightarrow' r$, and

(A2) there is no write w such that $\pi(r) \rightarrow' w \rightarrow' r$.

That is, the partially ordered set of actions can be linearized while respecting the logical read/write order. A shared variable is *atomic* if each system execution $\sigma = (\mathcal{A}, \rightarrow, \pi)$ of it is atomic.

1.4 The Problem to be Solved

Our goal is to implement an atomic wait-free shared variable V for n users $0, \ldots, n-1$, such that each user can perform both reads and writes. V is implemented using atomic variables $R_{i,j} (0 \leq i, j < n)$, for which user i is the only associated writer process and user j is the only associated reader process. Since user i is the only one who can write to variables $R_{i,0}, \ldots, R_{i,n-1}$, we say it *owns* these variables.

1.5 Specification

While the definition of atomicity is quite clear, we transform it into an equivalent specification, from which we can directly derive our first algorithm that implements V. Viz., partition the actions in $\mathcal{A}$ into subsets induced by write actions. Define the equivalance class of a write action w as $[w] = \{a : a = w$ or a is a read and $\pi(a) = w\}$. The precedence relation $\rightarrow$ on the actions in $\mathcal{A}$ induces a relation $\ll$ on the set of $[w]$'s as follows: $[w_1] \ll [w_2]$ iff $w_1 \neq w_2$ and there are $a_1 \in [w_1]$ and $a_2 \in [w_2]$ such that $a_1 \rightarrow a_2$. The following lemma comes from [1]:

Lemma 1 *(A1) and (A2) hold iff $\ll$ is acyclic and not $(r \longrightarrow \pi(r))$ for any read r.*

Proof. "If". If $\ll$ has no cycles then it can be extended to a total order $<$ on the set of $[w]$'s. Define $\longrightarrow'$ on $\mathcal{A}$ by:

1. if $[a] \neq [b]$ then $a \longrightarrow' b$ iff $[a] < [b]$
2. within each $[w]$ we can topologically sort the elements beginning with the write (since $r \not\longrightarrow \pi(r)$), so that $[w] = \{w, r_1, \ldots, r_k\}$ and $r_i \longrightarrow r_j$ implies $i < j$. Now put $w \longrightarrow' r_1 \longrightarrow' r_2 \longrightarrow' \cdots \longrightarrow' r_k$.

We claim that $\longrightarrow'$ is an extension of $\longrightarrow$. Assume that $a \longrightarrow b$. Then either $[a] \neq [b]$ and by the definition of $\ll$ we have that $[a] < [b]$ and thus $a \longrightarrow' b$, or $[a] = [b]$ and then we also have $a \longrightarrow' b$ because a is a write or $a = r_i$ and $b = r_j$ with $i < j$. Furthermore, $\longrightarrow'$ is a total order on $\mathcal{A}$ since the elements in each $[w]$ are totally ordered and the $[w]$ themselves are totally ordered by 1. Finally, (A1) and (A2) hold because $\pi(r)$ is the first write $\longrightarrow'$-preceding r.

"Only if". First assume $r \longrightarrow \pi(r)$ for some read r. Since $\longrightarrow'$ extends $\longrightarrow$, we also have $r \longrightarrow' \pi(r)$ which is in contradiction with (A1). So we have that $r \not\longrightarrow \pi(r)$ for any read r. Because of (A1) and (A2) each $[w]$ is a consecutive sequence of actions in the total order $\longrightarrow'$. It follows that the order $<$ on the $[w]$'s induced by $\longrightarrow'$ is total. If $[w_1] \ll [w_2]$ then there are $a_1 \in [w_1]$ and $a_2 \in [w_2]$ such that $a_1 \longrightarrow a_2$. Since $\longrightarrow'$ extends $\longrightarrow$, we also have $a_1 \longrightarrow' a_2$ and thus $[w_1] < [w_2]$. We see that $<$ is an extension of $\ll$. Since $<$ is acyclic, so must $\ll$ be. $\square$

This way we have found the specification that an atomic variable has to satisfy: for each of its system executions $\sigma = (\mathcal{A}, \longrightarrow, \pi)$

(S1) not $(r \longrightarrow \pi(r))$ for any read execution r, and

(S2) the induced relation $\ll$ has no cycles.

2 The Basic Algorithm

Our first approximation of the target algorithm captures the essence of the problem solution apart from the boundedness of the constituent shared variables. (In fact, this is essentially the "first solution" in [16], there presented with a different proof, and the matrix architecture is used in all later constructions [14, 7, 3, 6, 10]. This is the only multiwriter algorithm currently generally accepted as being correct — but it uses unbounded tags.) Let V be as in the problem description above. For most implementations of V condition (S1) is trivially satisfied, since violation of it would mean that a read execution returns a value before the write of it ever started. This condition will be trivially satisfied by all algorithms we consider, so we do not mention it any further. How can we satisfy (S2)? We proceed as follows. Let $(T, <)$ be a partially ordered set of *tags*. For each system execution $\sigma = (\mathcal{A}, \rightarrow, \pi)$, let $tag : \mathcal{A} \rightarrow T$ be a function such that

(T1) $tag(a) = tag(b)$ for $b \in [a]$,

(T2) if $a \rightarrow b$ then $tag(a) \leq tag(b)$, and

(T3) if $w_1 \neq w_2$ then $tag(w_1) \neq tag(w_2)$.

(T1) ensures that each $[w]$ has a unique tag. If $[w_1] \ll [w_2]$ then by definition $w_1 \neq w_2$ and there are $a \in [w_1]$ and $b \in [w_2]$ with $a \rightarrow b$, so that (T2) gives $tag(w_1) \leq tag(w_2)$ and (T3) gives $tag(w_1) \neq tag(w_2)$, which combine to $tag(w_1) < tag(w_2)$. It suffices to devise an algorithm such that for each system execution σ there is a function *tag* satisfying (T1), (T2), and (T3), in which case $\ll$ has no cycles and (S2) is satisfied.

Using unbounded tags, we can implement variable V as follows. (T1) will be satisfied by letting read-actions copy the tag of the value that they choose to return. (T2) will be satisfied by letting the users maximize over the tags that are visible in their actions. Different writers will use different tags by making the index of a writer part of its tag, i.e., each tag in T is a pair (t, i), where t is a natural number and i is the index $(0 \leq i < n)$ of the user that writes the tag first. The $<$-order on T is the total lexicographic order. Finally a writer will not use a tag that it has used before because it chooses its tag *greater* than the maximum visible tag. Thus, (T3) is also satisfied.

Figure 1 shows the basic algorithm. An atomic subvariable $R_{i,j}$ can be written to by user i in a statement `Write` $R_{i,j}$ `:= loc`, where `loc` is a local variable. Likewise, it can be read by user j in a statement `Read loc :=` $R_{i,j}$. Each $R_{i,j}$ contains the fields `value` and `tag`, while the two parts of `tag` will be referred to as the fields `timestamp` and `index`.

The algorithm is initialized by simply setting all fields of all local and subvariables to 0. This puts the system in a state which appears to have resulted from an initial write by user 0 of the value 0 with a tag of $(0, 0)$, followed by succesive reads of the other users $1, \ldots, n-1$, such that these all choose `max :=` 0.

Locally, each user i has an array `from[0]`,...,`from[n-1]` where `from[j]` is meant to hold a local copy of $R_{j,i}$. User i will use the local variable `from[i]` instead of $R_{i,i}$ (which is not really a *shared* variable), which makes the architecture an n by $n-1$ matrix. Note that a single protocol is given for both read and write actions, with branches to make the necessary distinctions. In all our algorithms, a `return` statement exits the execution of the protocol. The argument of the `return` statement is used by read actions as the return value, and is ignored by write actions. If the end of a protocol can only be reached by a write action, then we will refrain from using a `return` statement in the last line.

```
 1. for j:=0 to n-1 except i do Read from[j] := $R_{j,i}$
 2. select max such that $\forall$j (from[j].tag $\leq$ from[max].tag)
 3. if read_action then
 4.   from[i].value := from[max].value
 5.   from[i].tag := from[max].tag
 6. else if write_action then
 7.   from[i].value := newvalue
 8.   from[i].tag := (from[max].timestamp+1,i)
 9. endif
10. for j:=0 to n-1 except i do Write $R_{i,j}$ := from[i]
11. return from[i].value
```

Figure 1: Algorithm 0; protocol for user i

Lemma 2 *Algorithm 0 implements an atomic wait-free multi-user variable.*

Proof. (See also [1, 16].) Obviously Algorithm 0 is wait-free. We only have to argue atomicity. Let $\sigma = (\mathcal{A}, \longrightarrow, \pi)$ be a system execution according to Algorithm 0. (T1) is satisfied since a read action writes the same tag as the write action whose value it returns. The sequence of accesses to `from[i].tag` by user i is a repeated sequence of

in line 2 at least one read to determine the maximum tag,

in line 5/8 a write of a value which is at least the value it replaces,

in line 10 some reads to make the tag visible to all other users.

This proves the following claim.

Claim 1 *The sequence of tags, written by user i, is monotonically nondecreasing.*

Moreover, in a write action, the new timestamp of i is at least the previous timestamp plus 1, and the index equals i, so two different write actions, either by the same user or by two different users, must have different tags, thus satisfying (T3). Let $a \longrightarrow b$ be two actions by users i and j respectively. If $i = j$ then (T2) is satisfied by claim 1. Suppose $i \neq j$. Then action b will select its tag by maximizing over, among others, the tag that it read from $R_{i,j}$ which is at least the tag of a by claim 1, since action a finished before b started. Again (T2) holds and therefore (S2) is satisfied. Condition (S1) is satisfied trivially. □

3 Algorithm 1

The only problem with algorithm 0 is that $T = N \times \{0, \ldots, n-1\}$ is infinite. Through a series of transformations of the basic algorithm, we shall remove this inconvenience.

```
1. for j:=0 to n-1 except i do Read from[j] := $R_{j,i}$
2. select max such that $\forall$j (from[j].tag $\leq$ from[max].tag)
3. from[i].value := from[max].value
4. from[i].tag := from[max].tag
5. for j:=0 to n-1 except i do Write $R_{i,j}$ := from[i]
6. if read_action then return from[i].value
7. from[i].value := newvalue
8. from[i].tag := (from[max].timestamp+1,i)
9. for j:=0 to n-1 except i do Write $R_{i,j}$ := from[i]
```

Figure 2: Algorithm 1; protocol for user i

In order to motivate our first transformation, we introduce the following system execution scenario.

User 0 starts a write operation, but falls asleep after writing to $R_{0,1}$ in line 10. Then user 1 also starts a write operation, sees the new tag $(1, 0)$ of user 0, and falls asleep after writing to $R_{1,2}$ in line 10. This continues in the obvious manner until user $n-2$ writes to $R_{n-2,n-1}$ and also falls asleep. Clearly, none of the writing users have informed user $n-1$ of the ongoing activities, and as a result the maximum timestamp in column $n-2$ ($R_{0,n-2}, \ldots, R_{n-1,n-2}$) is approximately n greater than the maximum timestamp in column $n-1$. It will later prove useful to bound these differences. Algorithm 1 (figure 2) takes care of this by having the write actions also "propagate" the value and tag that they consider to be most recent. This way, the write protocol becomes an extension of the read protocol.

Since we will prove correctness of the final algorithm from scratch, we suffice to say that the correctness of algorithm 1 follows easily from that of algorithm 0, once we observe that claim 1 still holds.

4 Algorithm 2

It is in this algorithm that we introduce the concept of "shooting." Shooting is performed at the end of a write action and is targetted at the actions seen by the writer in the read phase. The idea is that an action starts by "healing" itself so that is is free from shots, then it does the usual copying of values to its `from[]` array, and then it checks whether it has been wounded by another user. This is the case if it has received a certain number of shots from that user. If wounded, then it will abort the protocol execution and return with a recent value written by the shooting user[2]. The number of shots is chosen so as to ensure that the wounded action completely overlaps the write action by the shooting user of the returned value.

[2]recall that by our convention a write action ignores the return value

```
 1. for j:=0 to n-1 except i do
 2.   Read tmp := $R_{j,i}$
 3.   from[i].die[j] := tmp.shoot[i]
 4. for j:=0 to n-1 except i do Write $R_{i,j}$ := from[i]
 5. for j:=0 to n-1 except i do Read from[j] := $R_{j,i}$
 6. for j:=0 to n-1 except i do
 7.   Read tmp := $R_{j,i}$
 8.   if tmp.shoot[i] - from[i].die[j] $\geq$ 3 then return tmp.previous
 9. select max such that $\forall$j (from[j].tag $\leq$ from[max].tag)
10. from[i].value := from[max].value
11. from[i].tag := from[max].tag
12. for j:=0 to n-1 except i do Write $R_{i,j}$ := from[i]
13. if read_action then return from[i].value
14. from[i].value := newvalue
15. from[i].tag := (from[max].timestamp+1,i)
16. for j:=0 to n-1 except i do
17.   if from[i].shoot[j] - from[j].die[i] < 5 then from[i].shoot[j] +:= 1
18. for j:=0 to n-1 except i do Write $R_{i,j}$ := from[i]
19. from[i].previous := from[i].value
```

Figure 3: Algorithm 2; protocol for user i

Looking at figure 3, we see that the implementation of the shooting adds quite a bit of complexity to the algorithm. All the old lines of Algorithm 1 are still there (lines 5,9,10,11,12,13,14,15,18), in the same order, making this algorithm an extension of Algorithm 1. In the added lines we can see some new fields in the registers. The field **previous** is used to hold the value of the last completed write (line 19). The shots from user j to user i are counted on **shoot[i]** of user j, while the healing takes place on **die[j]** of user i. The difference between these two counters will be a measure of the injuries suffered by user i's actions. Note that these counters are unbounded like the timestamps. We will later see that they can be easily bounded. However, doing so at this point would only complicate the algorithm and its discussion. The way it works is that at the start of a new action user i reads the **shoot[i]** entry of $R_{j,i}$. Then it sets its **die[j]** equal to that shoot count. During the reading phase (line 5) of i's action, user j may perform some write actions. At the end of each such action, user j checks whether the difference between **shoot[i]** and **die[j]** has reached 5 yet. If not then it shoots the current action of user i by incrementing (+:= 1) its shoot counter in line 17. This procedure clearly implies that the difference between a shoot counter and the corresponding die counter is always between 0 and 5 included. In this algorithm, we concentrate on the number of shots to wound an action (3). Namely, after the reading phase, user i will read the **shoot[i]** entry of $R_{j,i}$ again and compare the difference with its **die[j]** against 3. If the difference is smaller, then the normal course of actions is resumed. Otherwise, the action aborts, i.e., it exits execution of the protocol returning the value of a recently completed write action by user j.

We now give an informal motivation for the correctness of algorithm 2. The restriction of a system execution to the set of non-aborting actions behaves just like a system execution of algorithm 1, so we need only show that aborting actions do not violate atomicity. If some action aborts, then the write action that shoots it first must finish after the start of the former action, otherwise the shot would be cleared in the healing phase. Call the write action whose value it returns the *returned write*. Since that value is the **previous** field, another action must have started after the returned write to set that field. So the returned write finishes before the aborting action does. Also the returned write is so recent that it delivers at least the second shot so that it must start later than the aborting action. We conclude that the second-shot write action is completely overlapped, and in particular the spot where the returned write action occurs atomically. Now if the aborting action was a write action, then we can imagine that it occurred atomically just before that spot, while if it was a read action then we can imagine it having occurred atomically just after that spot. In both cases (A1) and (A2) are satisfied. Thus if we insert all aborting actions in this way, then we have shown the existence of a total atomic extension of $\longrightarrow$, hence the atomicity of the system execution.

5 Algorithm 3

In our quest for boundedness, we would like the actions to consider only recent tags. Since the shoot and die counters are visible to all users, they can serve the purpose of recognizing old actions. More precisely, if user i reads the $R_{j,i}$ quickly enough so as not to abort, and sees from[k].shoot[j] $\geq$ from[j].die[k] $+ 5$, then it will assume that from[j] is too old—at least older than from[k]. The value 5 is the "fatal" or "terminal" amount of shots and in the above case we say that *i sees j killed by k*, or simply *k has killed j*.

There is however one complication. Namely, if user i starts a new action, then its die counters— as visible in the $R_{i,j}$— still correspond to its previous action, and changing them as may be required in line 4 would invalidate this correspondence. The solution is to use separate die counters for an action in progress, and to cause them to come into effect as soon as a new value and tag have been chosen for propagation. In algorithm 3 (figure 4), we see that each shoot and die counter has been duplicated (as an array of length 2), and that an extra index is used in their addressing. There is also a new field by the name of ss (short for *shoot selector*), which is the index of the die counters corresponding to the values in the tag and value fields. The other set of counters (with index $1 -$ ss) is used for an action in progress which hasn't yet chosen a tag/value. The reason for also duplicating the shoot counters is that with two die counters, we might want to increase the shoot counter for one, while we might already be 5 ahead of the other[3]. The fields num, pnum,

[3]In fact, the two die counters can differ arbitrarily in case of repeated abortion.

```
1.  s := 1 - from[i].ss
2.  for j:=0 to n-1 except i do
3.    Read tmp := $R_{j,i}$
4.    from[i].die[j][s] := tmp.shoot[i][s]
5.  { from[i].num +:= 1 }
6.  for j:=0 to n-1 except i do Write $R_{i,j}$ := from[i]
7.  for j:=0 to n-1 except i do Read from[j] := $R_{j,i}$
8.  for j:=0 to n-1 except i do
9.    Read tmp := $R_{j,i}$
10.   if tmp.shoot[i][s] - from[i].die[j][s] $\geq$ 3 then return tmp.previous
11. L := { 0,...,n-1 }
12. select max $\in$ L such that $\forall$j $\in$ L (from[j].tag $\leq$ from[max].tag)
13. from[i].value := from[max].value
14. from[i].tag := from[max].tag
15. from[i].ss := s
16. { from[i].pnum := from[i].num }
17. for j:=0 to n-1 except i do Write $R_{i,j}$ := from[i]
18. if read_action then return from[i].value
19. from[i].value := newvalue
20. from[i].tag := (from[max].timestamp+1,i)
21. for j:=0 to n-1 except i and for s:=0 to 1 do
22.   if from[i].shoot[j][s] - from[j].die[i][s] < 5
23.   then from[i].shoot[j][s] +:= 1
24. { from[i].snum := from[i].num }
25. for j:=0 to n-1 except i do Write $R_{i,j}$ := from[i]
26. from[i].previous := from[i].value
```

Figure 4: Algorithm 3; protocol for user i

and snum serve no other purpose than simplifying the proofs. They number the actions of each user from 0 onwards. We place the auxiliary lines 5,16 and 24 between braces ({,}) to emphasize that they are not part of the algorithm itself. Algorithm 3 can be divided into 5 *phases*[4], defined as follows:

heal lines 1–6

read line 7

test lines 8–10

propagate lines 11–17

write lines 19–25

Lines 11 and 12 together are called the *maximization step*, and lines 21–23 the *shoot* phase.

[4]apart from the reader's point of return line 18

Claim 2 *For each of the fields* `shoot`, `die`, `timestamp`, `tag`, `num`, `pnum` *and* `snum` *in either a shared register or a local copy (the* `from[]` *array), it holds that the sequence of values is monotonically nondecreasing (in time).*

Proof. First consider the local variable `from[i]` of user i. Examination of the algorithm and the maximization step (line 12) in particular shows that the claim holds for the fields `num`, `pnum`, `snum`, `shoot`, `tag` and `timestamp` of this variable. It therefore holds for $R_{i,j}$ as well since that is only changed by assigning `from[i]` to it. The same applies to the `from[i]` variable of other users, and likewise to their `die[i]`, since that is only changed by assigning $R_{i,j}$.`shoot` to it. □

Before giving the proof of correctness of Algorithm 3, let us introduce some conventions and notations to help state the proofs.

Definition 1 Let $a \in \mathcal{A}$ be an action by user i. Define $sr(a)$ and $fr(a)$ as the start time and finish time of the read phase respectively, such that

- all accesses in the heal phase occur between $s(a)$ and $sr(a)$,
- all accesses in the read phase occur between $sr(a)$ and $fr(a)$ and
- all accesses in the propagate and write phase occur between $fr(a)$ and $f(a)$.

Definition 2 Let $a \in \mathcal{A}$ be an action by user i, `lnr` a line number, and `loc` a local variable of user i. We define `loc`(a.`lnr`) as the value of that variable at the time a completes line `lnr` of its protocol. If `loc` is a field of `from[i]` then we may omit the `from[i].` part and just give the field name. The line number may also be omitted in which case the value at the end of the protocol execution of a is meant.

With a few exceptions, `loc`(a) is the single value assigned to `loc` during the protocol execution of a. When referring to values as they are at the end of the maximization step—which is often necessary in case the `from` subscript may equal i—we use the notation `loc`(a.12).

Lemma 3 *Let a be a non-aborting action by user i and b an action by user j with* `num`(b) = `from[j].pnum`(a.12). *Define js =* `from[j].ss`(a.12). *Then*

1. *b does not abort*
2. `ss`(b) $= js$
3. *for any $k \in \{0, \ldots, n-1\}$,* `die[k][js]`(b) = `from[j].die[k][js]`(a.12)
4. `tag`(b) $\geq$ `from[j].tag`(a.12)

Proof. The case $i = j$ is straightforward, so we assume $i \neq j$[5]. If b aborts, then it doesn't execute line 16 and therefore $\mathrm{num}(b)$ is never written to $R_{j,i}$.pnum. This contradicts from[j].pnum(a.12) = num(b), so b doesn't abort. Furthermore, the value of $R_{j,i}$.pnum remains num(b) until the first non-aborting action by user j, succeeding b, writes its num in that subvariable—which it does in line 17. At that moment pnum changes again. And only then can $R_{j,i}$ss or $R_{j,i}$die[k][js] change (as line 1 shows). Given the fact that action a observes the pnum of b, it can only be that the observed ss and die[k][js] equal those of b. This proves js = ss(b) and that for any k, die[k][js](b) = from[j].die[k][js](a.12). Finally, for a to see a tag greater than that of b, a change in pnum is also required, contradictory to the definition of b. □

Lemma 3 provides a convenient way to relate actions and will be used whenever we introduce an action by its num.

5.1 Correctness of Algorithm 3

Theorem 2 *Let $\sigma = (\mathcal{A}, \longrightarrow, \pi)$ be a system execution according to Algorithm 3. Then there is a function tag() satisfying (T1), (T2) and (T3).*

Proof. (T1) states that $tag(r) = tag(\pi(r))$ must hold for all read actions r. Let a be an action by user i. We will satisfy (T1) by defining $tag : \mathcal{A} \longrightarrow R$ as follows. If a doesn't abort, then define $tag(a) = n \times$ timestamp(a) + index(a). For convenience we will sometimes forget the distinction between $tag(a)$ and tag(a). If a is an aborting read action, then define $tag(a) = tag(w)$, where w is the non-aborting write of the return-value tmp.previous(a). Clearly this definition of tag satisfies (T1). Note that tag remains to be defined on aborting write actions. This will be done (more conveniently) in the next lemma.

Lemma 4 *If an action a by user i aborts, then there exists a non-aborting write action w, such that $s(a) < s(w) < f(w) < f(a)$ and $\lceil tag(a) \rceil = tag(w)$.*

Proof. Define j = j(a) as the user by which a sees itself wounded[6]. Let w be the last non-aborting write action by user j with num(w) < tmp.num(a). Then obviously $f(w) < f(a)$. Examination of Algorithm 3 shows that value(w) = tmp.previous(a). From the definition of w and the abortion of a follows shoot[i](w) $\geq$ tmp.shoot[i](a) $- 1 \geq$ die[j](a) + 2. Therefore, the previous action u of user j has shoot[i](u) $\geq$ die[j](a) + 1, so $\neg(u \longrightarrow a)$. This proves $s(a) < s(w)$. If a is a read action then $\lceil tag(a) \rceil = tag(a) = tag(w)$. Otherwise, a is an aborting write action. We complete the definition of tag with $tag(a) = tag(w) - \epsilon_a$, where $0 < \epsilon_a < 1$ is a small positive real number unique to a. □

[5]The .12 extension is used to generalize the lemma to the case $i = j$. It can be omitted for $i \neq j$.

[6]recall that a ends execution of the protocol in lines 8–10.

(T2) states that if $a \rightarrow b$ then $tag(a) \leq tag(b)$. Let $a \rightarrow b$ be actions by users i and j respectively.

Case 1: a does not abort. Suppose b doesn't abort either. Then

$$tag(b) = \mathbf{tag}(b) \geq \texttt{from[max].}\mathbf{tag}(b.12)$$

$$\geq \texttt{from[i].}\mathbf{tag}(b.12) \geq \mathbf{tag}(a) = tag(a).$$

The last inequality is based on claim 2, and in case $i \neq j$ also on the fact that a writes $tag(a)$ to $R_{i,j}$.

Now suppose b aborts. By lemma 4 there is a non-aborting write action w_b with $tag(b) > tag(w_b) - 1$ and $a \rightarrow w_b$. Then

$$tag(w_b) - 1 = \mathbf{tag}(w_b) - 1 \geq \texttt{from[max].}\mathbf{tag}(w_b.12)$$

$$\geq \texttt{from[i].}\mathbf{tag}(w_b.12) \geq \mathbf{tag}(a) = tag(a).$$

The first inequality follows from the increment of the timestamp in line 20 of the write protocol.

Case 2: a aborts. Then by lemma 4 there is a non-aborting w_a with $tag(a) \leq tag(w_a)$ and $w_a \rightarrow b$. Hence this case reduces to the former.

(T3) states that if $w_1 \neq w_2$ then $tag(w_1) \neq tag(w_2)$. Let i, j be the users executing w_1 and w_2 respectively. If either w_1 or w_2 aborts, then either $tag(w_1)$ or $tag(w_2)$ has a unique fractional part, hence (T3) holds. Now assume that neither one aborts. Then $tag(w_1) = i \pmod n$, while $tag(w_2) = j \pmod n$. Clearly, (T3) holds in case $i \neq j$. In case $i = j$, assume w.l.o.g. that $w_1 \rightarrow w_2$. Then

$$tag(w_2) = \mathbf{tag}(w_2) > \texttt{from[max].}\mathbf{tag}(w_2.12)$$

$$\geq \texttt{from[i].}\mathbf{tag}(w_2.12) \geq \mathbf{tag}(w_1) = tag(w_1).$$

The last inequality follows from claim 2. □

5.2 An important Modification to Algorithm 3

We will now show how L as set in line 11 can be restricted without affecting the choice of `max`. Lemma 5 implies that an index with maximum tag cannot be killed.

Lemma 5 *Let $\sigma = (A, \rightarrow, \pi)$ be a system execution according to Algorithm 3, a a non-aborting action by user i, and $j, k \in L$ two indices. Define js =* `from[j].ss`$(a.12)$. *If*

$$\texttt{from[k].shoot[j][js]}(a.12) - \texttt{from[j].die[k][js]}(a.12) \geq 5$$

(a sees j killed by k), then

$$\texttt{from[k].}\mathbf{tag}(a.12) > \texttt{from[j].}\mathbf{tag}(a.12)$$

Proof. Let b be the action by user j with $\mathtt{num}(b) = \mathtt{from[j].pnum}(a.12)$. Let $w_0 \to \cdots \to w_5$ be non-aborting write actions by user k such that

$$\mathtt{shoot[j][js]}(w_m) = \mathtt{die[k][js]}(b) + m,$$

where $0 \leq m \leq 5$, and $\mathtt{tag}(w_5) \leq \mathtt{from[k].tag}(a.12)$.

Since by lemma 3, b doesn't abort, it is not the case that $f(w_3) < fr(b)$. Therefore $fr(b) < f(w_3) < s(w_4)$.

Since $\mathtt{tag}(w_5) \geq (\mathtt{timestamp}(w_4) + 1, k)$, and application of lemma 3 yields

$$(\mathtt{from[max].timestamp}(b.12) + 1, j) \geq \mathtt{tag}(b) \geq \mathtt{from[j].tag}(a.12),$$

we can prove lemma 5 by showing that

$$(\mathtt{timestamp}(w_4) + 1, k) > (\mathtt{from[max].timestamp}(b.12) + 1, j).$$

We examine two cases:

$j < k$ Define $id = \mathtt{from[max].index}(b.12)$. Because of the propagate phase, $\mathtt{from[max].timestamp}(b.12) - 1$ is written in $R_{id,k}$ before $fr(b)$. So

$$(\mathtt{timestamp}(w_4) + 1, k) \geq (\mathtt{from[max].timestamp}(b.12) + 1, k)$$
$$> (\mathtt{from[max].timestamp}(b.12) + 1, j).$$

$k < j$ From the ordering of the **for** loops in lines 17 and 25, we deduce that $\mathtt{from[max].timestamp}(b.12)$ is written in $R_{\mathtt{max}(b),k}$ before $fr(b)$. So

$$(\mathtt{timestamp}(w_4) + 1, k) \geq (\mathtt{from[max].timestamp}(b.12) + 2, k)$$
$$> (\mathtt{from[max].timestamp}(b.12) + 1, j).$$

□

Lemma 5 is very important, since it shows that we can replace line 11 in Algorithm 3 with

```
11'.  L = {j | ∀ k≠j from[k].shoot[j][from[j].ss]-from[j].die[k][from[j].ss] < 5}
```

Intuitively, we restrict the set from which to choose max to the *alive* tags. The resulting algorithm will be referred to as Algorithm 4.

6 Final Algorithm

In this section we will show how to get rid of the unbounded tags and counters in Algorithm 4. Additionally, the control-bit complexity will be minimized by removing from $R_{i,j}$ any information not relevant to user j.

Due to the behaviour of shoot and die counters, the perceived differences in lines 10 and 22 are in the range from 0 to 5 inclusive. This is not the case in line 11' where the two comparands have been acquired at different times. We will show that these differences are bounded nevertheless.

Suppose user i is active in the read phase of action a. If ρ_j and ρ_k with $j < k$ are the atomic reads from $R_{j,i}$ and $R_{k,i}$ respectively, then other users may be executing arbitrarily many actions between ρ_j and ρ_k. Generally, this has the effect that the values obtained from $R_{k,i}$ in ρ_k are greater then they were at the time of ρ_j. Thus, k appears to have higher tags, higher shoot counters, and higher die counters. Since read actions do not introduce any higher values than those already in existence, this effect must be caused by write actions. We have however introduced a shooting mechanism which makes actions abort if they overlap—in their read phase—a certain number of write actions by the same user. Furthermore, those aborting actions do not even get to line 11'. By bounding the number of overlapped write actions, we can also bound the apparent increase in k's values. The next two sections discuss the differences between shoot and die counters and after that comes a discussion of timestamp differences.

6.1 Upper Bound on Perceived Differences

Lemma 6 *Let a be a non-aborting action by user i, and $0 \le j, k < n$. Define $js =$ `from[j].ss`$(a.12)$. Then*

$$\texttt{from[k].shoot[j][js]}(a.12) - \texttt{from[j].die[k][js]}(a.12) \le 8$$

Proof. Let b be the action by user j with $\texttt{num}(b) = \texttt{from[j].pnum}(a.12)$. Let $w_0 \to \cdots \to w_9$ be non-aborting write actions by user k such that

$$\texttt{shoot[j][js]}(w_m) = \texttt{die[k][js]}(b) + m,$$

where $0 \le m \le 9$. Since

$$\texttt{die[k][js]}(b) + 6 = \texttt{shoot[j][js]}(w_6) \le \texttt{from[j].die[k][js]}(w_6) + 5,$$

we have that

$$\texttt{from[j].die[k][js]}(w_6) > \texttt{die[k][js]}(b).$$

Therefore, if we let d be the first action by user j with $\texttt{die[k][js]}(d) > \texttt{die[k][js]}(b)$, then $f(w_6) > s(d)$. Furthermore, there must exist a non-aborting action c, with $b \to c \to d$ and $\texttt{ss}(c) = 1 - \texttt{ss}(b)$[7]. Since $\texttt{num}(c) >$

[7]See also the proof of lemma 3.

$\mathrm{num}(b) = \texttt{from[j]}.\mathrm{pnum}(a.12)$, and c writes the latter to $R_{j,i}$.pnum, we have $\neg(f(c) < sr(a))$. This leads to the following ordering of events:

$$sr(a) < f(c) < s(d) < f(w_6) < s(w_7).$$

As a consequence (with $is = \texttt{ss}(a)$),

$$\texttt{from[i].die[k][is]}(w_7) \geq \texttt{die[k][is]}(a),$$

so w_7 and the following write actions by user k will take shots at a. Formally, $\texttt{shoot[i][is]}(w_m) \geq \texttt{die[k][is]}(a) + m - 6$, $7 \leq m \leq 9$. We now use the fact that a does not abort to conclude that $\texttt{from[k].snum}(a.12) < \mathrm{num}(w_9)$, hence

$$\texttt{from[k].shoot[j][js]}(a.12) < \texttt{shoot[j][js]}(w_9) =$$

$$\texttt{die[k][js]}(b) + 9 = \texttt{from[j].die[k][js]}(a.12) + 9.$$

The last equality follows of course from lemma 3. □

6.2 Lower Bound on Perceived Differences

We stick to the convention that k shoots at j, and consider the case $k < j$, and therefore $\rho_k \rightarrow \rho_j$. It is possible that between these two atomic reads, user k performs some writes, and user j catches up with a read action. It can then be the case that the die counter read in ρ_j is actually greater than the shoot counter read in ρ_k!

Lemma 7 *Let a be a non-aborting action by user i, and $0 \leq j, k < n$. Define $js = \texttt{from[j].ss}(a.12)$. Then*

$$\texttt{from[k].shoot[j][js]}(a.12) - \texttt{from[j].die[k][js]}(a.12) \geq -4$$

Proof. Let $w_0 \rightarrow \cdots \rightarrow w_5$ be non-aborting write actions by user k such that

$$\texttt{shoot[j][js]}(w_m) = \texttt{from[k]shoot[j][js]}(a.12) + m,$$

where $0 \leq m \leq 5$. From this definition it follows that $\neg(f(w_1) < sr(a))$. As a consequence, w_2 and the following write actions by user k will take shots at a. Formally (with $is = \texttt{ss}(a)$),

$$\texttt{shoot[i][is]}(w_m) \geq \texttt{die[k][is]}(a) + m - 1,$$

for $0 \leq m \leq 5$. We use the fact that a does not abort to conclude that $fr(a) < f(w_4)$. Let b be the action by user j with $\mathrm{num}(b) = \texttt{from[j]}.\mathrm{pnum}(a.12)$. Then by lemma 3 and because $sr(b) < fr(a) < f(w_4) < s(w_5)$,

$$\texttt{from[j].die[k][js]}(a.12) = \texttt{die[k][js]}(b) \leq$$

$$\texttt{shoot[j][js]}(w_4) = \texttt{from[k]shoot[j][js]}(a.12) + 4$$

The inequality follows from the observation that b fixes its die counter before w_5 starts—therefore it cannot be greater than the shoot counter of w_4. □

We observe that the result of any comparison between a shoot counter and the corresponding die counter always lies in the range $-4, \ldots, 8$ inclusive. Since the only purpose served by these counters is comparison, we might as well store them modulo $8 + 1 - (-4) = 13$. For the algorithm, this only involves performing the shoot counter increment modulo 13 and mapping all differences between shoot and die counters to the above range. This mapping can be done in the comparison by treating $9, 10, 11, 12$ as -4, -3, -2, -1. Instead of counter, the name *ring* now seems more appropriate. There are shoot rings and die rings, and the next section will introduce another member in the ring-family.

6.3 Range of Alive Timestamps

Having restricted L in Algorithm 4 to the set of alive tags, we would expect the tags—and therefore the timestamps—to be relatively close to each other. This is formalized in the next lemma.

Lemma 8 *Let a be a non-aborting action by user i, and $0 \leq j, l < n$. Define $js =$ `from[j].ss`$(a.12)$. If*

$$\texttt{from[l].timestamp}(a.12) - \texttt{from[j].timestamp}(a.12) > 9n$$

then there exists a $k, 0 \leq k < n$ such that

$$\texttt{from[k].shoot[j][js]}(a.12) - \texttt{from[j].die[k][js]}(a.12) \geq 5.$$

Proof. Let b be the action by user j with $\texttt{num}(b) =$ `from[j].pnum`$(a.12)$, so that `from[j].timestamp`$(a.12) \geq$ `from[max].timestamp`$(b.12)$. More than $9n$ non-aborting write actions—starting before $fr(a)$—must have a timestamp greater than `from[max].timestamp`$(b.12)$. Therefore all these actions finish after $sr(b)$. Since each user can start at most one such action before $sr(b)$, no more than n of the over $9n$ write actions start before $sr(b)$. Therefore more than $8n$ of the write actions start after $sr(b)$. Let k be a user which executed at least 9 of those write actions. Rename these as $u_1 \rightarrow \cdots \rightarrow u_9$. Then (because $sr(b) < s(u_1)$) these actions will shoot at b, as in

$$\texttt{shoot[j][js]}(u_m) \geq \texttt{die[k][js]}(b) + m,$$

for $1 \leq m \leq 5$. Since $f(u_8) < s(u_9) < fr(a)$, it cannot be that $sr(a) < s(u_6)$, because in that case u_6, u_7 and u_8 would all shoot a and would thus cause a to abort; a contradiction. So then $f(u_5) < s(u_6) < sr(a)$, and therefore

$$\texttt{from[k].shoot[j][js]}(a.12) \geq \texttt{shoot[j][js]}(u_5)$$

```
 1. s := 1 - from[i].ss
 2. for j:=0 to n-1 except i do
 3.   Read tmp := $R_{j,i}$
 4.   from[i].die[j][s] := tmp.shoot[i][s]
 5. for j:=0 to n-1 except i do Write $R_{i,j}$ := from[i]
 6. for j:=0 to n-1 except i do Read from[j] := $R_{j,i}$
 7. for j:=0 to n-1 except i do
 8.   Read tmp := $R_{j,i}$
 9.   if (tmp.shoot[i][s] - from[i].die[j][s]) mod 13 $\geq$ 3
10.   then return tmp.previous
11. L := { j | $\forall$ k$\neq$j (from[k].shoot[j][from[j].ss]
      - from[j].die[k][from[j].ss]) mod 13 $\notin$ {5,...,8}) }
12. select max $\in$ L such that $\forall$ j $\in$ L
13.   dts $\leq$ 9n $\wedge$ (dts > 0 $\vee$ max > j), where
14.   dts=(from[max].timestamp - from[j].timestamp) mod (18n+1)
15. from[i].value := from[max].value
16. from[i].tag := from[max].tag
17. from[i].ss := s
18. for j:=0 to n-1 except i do Write $R_{i,j}$ := from[i]
19. if read_action then return from[i].value
20. from[i].value := newvalue
21. from[i].tag := ((from[max].timestamp+1) mod (18n+1), i)
22. for j:=0 to n-1 except i and s $\in$ {0,1} do
23.   if (from[i].shoot[j][s] - from[j].die[i][s]) mod 13 < 5
24.   then from[i].shoot[j][s] := (from[i].shoot[j][s]+1) mod 13
25. for j:=0 to n-1 except i do Write $R_{i,j}$ := from[i]
26. from[i].previous := from[i].value
```

Figure 5: Final Algorithm; protocol for user i

$$\geq \texttt{die[k][js]}(b) + 5 = \texttt{from[j].die[k][js]}(a.12) + 5.$$

The last equality follows again from lemma 3. □

By lemma 8, the result of comparing two alive timestamps in Algorithm 4 lies in the range $-9n, \ldots, 9n$ inclusive. That being the only use of timestamps, we might as well use all timestamps modulo $18n + 1$, without affecting the behaviour of the algorithm. With these *timestamp rings*, we have bounded all variables used by the algorithm. The final algorithm, incorporating all three kinds of ring, is worth a picture (fig 5).

We may note that user j makes no use of the fields die[$\neq j$][1-ss] in $R_{i,j}$. We exploit this by storing only a single die[] array in $R_{i,j}$, where the second index is understood to be the $R_{i,j}$.ss, and in addition the single die ring die[j][1-ss]. Table 2 shows all the fields in $R_{3,1}$ in the case of 5 users.

shoot[0][0] shoot[0][1]	shoot[1][0] shoot[1][1]	shoot[2][0] shoot[2][1]	shoot[4][0] shoot[4][1]
value previous	timestamp index	ss	
die[0][ss]	die[1][1-ss] die[1][ss]	die[2][ss]	die[4][ss]

Table 2: the fields of $R_{3,1}$ with 5 users

7 Subproblems and Implementation

The final algorithm implements a multi-user variable with single-reader variables. If we are given *multi-reader* variables as the basis of a multi-user implementation, then we can make some simplifications. The interested reader is referred to appendix A. Another projection of the algorithm implements a multi-reader variable from single-reader ones. See appendix B for details.

Some of the algorithms presented here have been implemented and tested. See appendix C[8] for the results.

8 Conclusions

We have proven that Algorithm 5, by equivalence with Algorithm 3, correctly implements an atomic multi-user variable without waiting. The number of subvariable accesses in an operation execution is at most $6(n-1)$ in the case of a write action and at most $5(n-1)$ in the case of a read action. This proves the $O(n)$ subvariable accesses or "time complexity" of theorem 1. The number of bits in each subvariable $R_{i,j}$ equals $2b+(3n-2)\log 13+\log(18n+1)+\log n+1$, with b the number of bits ("width") of the shared variable V. This proves the $O(n)$ control bits or "space complexity" of theorem 1, and thereby completes its proof. Unfortunately, the extra value `previous` may put a heavy burden on the size of the subvariables if $b >> n$. This cannot be helped since the whole solution relies on the principle of abortion and aborting actions are required to return the value of a completed write.

We would like to make some final remarks about the possibility of parallelizing the algorithm. The order of the for loops in Algorithm 5 is only important in lines 18 and 25. All other for loops could be replaced by a "`for all j in`" construct which means that the different instances of the loop body can be executed in parallel. By increasing the lethal number of shots from 5 to 6, the above two loops can also be relaxed. This would enlarge the range of perceived `shoot - die` differences to $-4,\ldots,9$ hence a 14-valued shoot/die ring (instead of 13).

[8]Incidentally, the appendix name equals the name of the programming language we used.

The size of the timestamp ring should accordingly be increased from $18n + 1$ to $20n + 1$. These numbers follow from careful examination of the lemmas 6, 7, and 8 in the light of the incremented lethal number of shots. We end the paper with the conclusion that a parallellized version of the algorithm can be made to run in $O(1)$ time complexity and the same $(O(n))$ space complexity.

9 Acknowledgements

We thank Amos Israeli for many fruitful discussions.

```
s := 1 - from[i].ss
for j:=0 to n-1 except i do
    Read tmp := $R_j$
    from[i].die[j][s] := tmp.shoot[i][s]
Write $R_i$ := from[i]
for j:=0 to n-1 except i do Read from[j] := $R_j$
for j:=0 to n-1 except i do
    Read tmp := $R_j$
    if (tmp.shoot[i][s] - from[i].die[j][s]) mod 9 $\geq$ 2
    then return tmp.value
L := { j | $\forall$ k$\neq$j (from[k].shoot[j][from[j].ss]
 - from[j].die[k][from[j].ss]) mod 9 $\notin$ {4,5,6}) }
select max $\in$ L such that $\forall$ j $\in$ L
    dts $\leq$ 9n $\wedge$ (dts > 0 $\vee$ max > j), where
    dts=(from[max].timestamp - from[j].timestamp) mod (12n+1)
if read_action then return from[max].value
from[i].value := newvalue
from[i].tag := ((from[max].timestamp+1) mod (12n+1), i)
from[i].ss := s
for j:=0 to n-1 except i and s $\in$ {0,1} do
    if (from[i].shoot[j][s] - from[j].die[i][s]) mod 9 < 4
    then from[i].shoot[j][s] := (from[i].shoot[j][s]+1) mod 9
Write $R_i$ := from[i]
```

Figure 6: Algorithm 6; protocol for user i

A From Multi-reader to Multi-user

A restriction of our original goal corresponds to the problem that [16] and [11] unsuccesfully tried to solve. Viz., to implement an atomic wait-free n-user shared variable V using atomic multi-reader variables $R_i (0 \leq i < n)$ for which user i is the single writer and the other $n-1$ users are the readers.

To implement V, collapse row $R_{i,0}, \ldots, R_{i,n-1}$ in Algorithms 0–5 to the single multi-reader variable R_i owned by user i. So the write loops turn into a single atomic write. Each variable $R_{j,i}$ read is replaced by Rj. The **previous** field is no longer needed, since the value of a write action becomes visible to all users at the same time, thus making it suitable for return by an aborting action. The complete propagate phase can be skipped for similar reasons. The wounding number of shots can be reduced from 3 to 2, since the write action firing the second shot must have finished by the time the shot is noticed. The lethal (killing) number of shots can then be reduced to 4. The affected ring-sizes are 9 for the shoot/die rings and $12n+1$ for the timestamp rings. Figure 6 shows the resulting algorithm, given without a proof of correctness. This solution uses $O(n)$ control bits per variable R_i and $O(n)$ atomic accesses of subvariables per read or write action on V.

```
from.value := newvalue
from.timestamp := (from.timestamp+1) mod 8
for j:=0 to n-1 do
  Read tmp := $R_{j,n}$
  for s ∈ {0,1} do
  if (from.shoot[j][s] - tmp.die[s]) mod 8 < 3
  then from.shoot[j][s] := (from.shoot[j][s]+1) mod 8
for j:=0 to n-1 do Write $R_{n,j}$ := from
from.previous := from.value
```

Figure 7: Algorithm 7; protocol for writer (user n)

B From Single-reader to Multi-reader

Another restriction of our original goal above corresponds to the problem that was attacked in [14, 7, 3, 10, 6]. Viz., to implement an atomic wait-free multi-reader shared variable V with n readers $0, \ldots, n-1$ and a single writer n. V is implemented using atomic single-reader variables $R_{i,j}, (0 \leq i, j \leq n)$ for which user i is the single associated writer and user j is the single associated reader. We show how to implement V using $O(n)$ control bits in each variable $R_{n,j}$ owned by the writer, and $O(1)$ control bits in each variable $R_{i,j}, (0 \leq i < n)$ owned by the readers.

With a single writer, there obviously is no need for an index part in the tag, none of the readers need a `shoot` array, and the `die[j][s]` array of a reader collapses into `die[s]`. Also the writer doesn't need a `die` array and therefore no `ss`. While the wounding number of shots remains 3, the killing number of shots can be reduced to 3. Because the writer will be last in the reader's read phase, the observed die counters (timestamps) are at most 1 greater than the corresponding shoot counters (resp. timestamp) of the writer. As a result, the size of the shoot/die rings can be reduced to 8. Analysis also shows that alive readers are perceived to be at most 6 timestamps behind the writer, so a timestamp ring of size 8 suffices. There are separate protocols for the writer (fig 7) and the readers (fig 8), as to reflect the functional difference. Note that the writer has no propagate phase—there are no other writers that could have more recent values.

```
s := 1 - from[i].ss
Read from[n] := $R_{n,i}$
from[i].die[s] := from[n].shoot[i][s]
Write $R_{i,n}$ := from[i]
for j:=0 to n except i do Read from[j] := $R_{j,i}$
if (from[n].shoot[i][s] - from[i].die[s]) mod 8 $\geq$ 3
then return from[n].previous
max := n
for j:=0 to n-1 except i do
  if from[j].timestamp = (from[n].timestamp+1) mod 8 $\wedge$
  (from[n].shoot[j][from[j].ss]-from[j].die[from[j].ss]) mod 8 $\notin$ {3,...,6})
  then max := j
from[i].value := from[max].value
from[i].timestamp := from[max].timestamp
from[i].ss := s
for j:=0 to n-1 except i do Write $R_{i,j}$ := from[i]
return from[i].value
```

Figure 8: Algorithm 7; protocol for reader i

C Simulation of the Algorithm

Both Algorithm 3 and Algorithm 5 have been implemented and a program was written which simulates (pseudo-) randomly interleaved system executions of both algorithms in parallel.

An explanation of the interleaving process follows. First a distribution of *sleeptimes* was fixed. A sleeptime is a number of steps which a user has to wait before it may continue the execution of a protocol. Initially all users start *awake*, i.e., with a sleeptime of zero. In general, if more than one user is awake, then one such user is selected at random and a new, positive sleeptime is chosen from the sleeptimes distribution. This procedure is repeated until a single user remains awake. In that case the minimum of the sleeptimes of the other users is determined and taken as the number of steps to run the remaining user. A step is defined as a part of the protocol involving exactly one primitive register access and any local computations—this reflects the atomicity of the constituent registers. When the desired number of steps is completed, it is subtracted from the sleeptimes of the other users and the whole process repeats. This sleeptime algorithm helps to find counter examples in which one user is required to sleep while most of the other ones repeatedly run.

By testing whether actions have the same behaviour under both protocols, the program empirically tests the correctness of Algorithm 5 relative to Algorithm 3. While the latter uses unbounded counters and tags, the former uses modulos and a restricted "alive" set to choose max from. The simulator allows the setting of the following parameters:

- number of users (n)
- lower bound of the range of shoot/die ring values (lb)
- upper bound of the range of shoot/die ring values (ub)
- number of shots to wound an action (wd)
- number of shots to kill an action (kl)
- size of the ring of timestamps (ts)
- percentage of write actions (wp)
- number of steps simulated with same sleeptime distribution (nd)
- size of the sleeptime distribution (ds)
- maximum sleeptime (ms)

Given the last two parameters, the logarithms of the ds sleeptimes were chosen randomly from the uniform distribution $[0, \log ms]$. After each nd simulated

lb	*ub*	*kl*	*ts*	*ms*	steps
−4	8	5	55	64	
−3					
−2				32	10^6
−1				32	10^5
	7				
	6				10^6
	5				10^6
		4		48	10^6
		3		48	10^4
			21		
			17		10^5
			13		10^5
			9		10^3

Table 3: simulation results.

steps a new sleeptime distribution is chosen. Sooner or later we will find a distribution that is appropriate for the search of a counter example. This means that *nd* should be taken neither too small nor too large; any value between 10^3 and 10^6 seems reasonable. Curiously, the best value of *wp* for finding counter examples proved to be 100. We refer to table 3 for some of the results obtained. The last column shows the order of the average number of steps executed before the simulator runs into a discrepancy between the two algorithms. The top row shows the default values and the blank last entry shows that the simulator kept running for millions of steps without ever detecting a failure. A blank entry in any column but the last refers to the default value. Throughout the testing done for compiling table 3, we fixed $n = 3$, $wd = 3$, $wp = 100$, $nd = 100000$, and $ds = 6$. The reason for the fixed number of users is that the set of counter examples found with $n = 4$ was a proper subset of those found with $n = 3$.

The counter examples which can be shown to exist for $lb = -3$ and $ub = 7$ proved to be too hard to find for the simulator within some 10^7 simulated steps. The bounds of the shoot/die rings are optimal in the sense that they are sufficient to preserve correctness of the algorithm, while tightening either of them gives rise to a counter example. The size of the timestamp ring is larger than necessary. Analysis leads us to believe that a size of $18n - 27$ is optimal, but a proof of that claim (if it exists) would probably at least double the length of this article!

While the use of a simulator may seem of questionable value in supporting the correctness proofs of the algorithms, it has proven to be of great assistance in the development of both the algorithms and their formal proofs. Several alternative implementations of the shooting construct have been tried out, and some were refuted by the simulator much faster than could have been done manually.

References

[1] B. Awerbuch, L. Kirousis, E. Kranakis, P.M.B. Vitányi, *A proof technique for register atomicity*, Proc. 8th Conference on Foundations of Software Technology & Theoretical Computer Science, Lecture Notes in Computer Science, vol. 338, pp. 286–303, Springer Verlag, 1988.

[2] B. Bloom, *Constructing Two-writer Atomic Registers*, IEEE Transactions on Computers, vol. 37, pp. 1506–1514, 1988.

[3] J.E. Burns and G.L. Peterson, *Constructing Multi-reader Atomic Values From Nonatomic Values*, Proc. 6th ACM Symposium on Principles of Distributed Computing, pp. 222–231, 1987.

[4] D. Dolev and N. Shavit, *Bounded Concurrent Time-Stamp Systems Are Constructible*, Proc. 21th ACM Symposium on Theory of Computing, 1989. (to appear)

[5] M.P. Herlihy, *Impossibility and Universality Results for Wait-Free Synchronization*, Proc. 7th ACM Symposium on Principles of Distributed Computing, 1988.

[6] A. Israeli and M. Li, *Bounded Time-Stamps*, Proc. 28th IEEE Symposium on Foundations of Computer Science, pp. 371–382, 1987.

[7] L.M. Kirousis, E. Kranakis, P.M.B. Vitányi, *Atomic Multireader Register*, Proc. 2nd International Workshop on Distributed Computing, Amsterdam, J. van Leeuwen (ed.), Springer Verlag Lecture Notes in Computer Science, vol. 312, pp. 278–296, July 1987.

[8] L. Lamport, *On Interprocess Communication Parts I and II*, Distributed Computing, vol. 1, pp. 77–101, 1986.

[9] N. Lynch and M. Tuttle, *Hierarchical correctness proofs for distributed algorithms*, Proc. 6th ACM Symposium on Principles of Distributed Computing, 1987.

[10] R. Newman-Wolfe, *A Protocol for Wait-Free, Atomic, Multi-Reader Shared Variables*, Proc. 6th ACM Symposium on Principles of Distributed Computing, pp. 232–248, 1987.

[11] G.L. Peterson and J.E. Burns, *Concurrent reading while writing II: the multiwriter case*, Proc. 28th IEEE Symposium on Foundations of Computer Science, pp. 383–392, 1987.

[12] G.L. Peterson, *Concurrent reading while writing*, ACM Transactions on Programming Languages and Systems, vol. 5, No. 1, pp. 46–55, 1983.

[13] R. Schaffer, *On the correctness of atomic multi-writer registers*, Technical Report MIT/LCS/TM-364, MIT lab. for Computer Science, June 1988.

[14] A.K. Singh, J.H. Anderson, M.G. Gouda, *The Elusive Atomic Register Revisited*, Proc. 6th ACM Symposium on Principles of Distributed Computing, pp. 206–221, 1987.

[15] K. Vidyasankar, *Converting Lamport's Regular Register to an atomic register*, Information Processing Letters, vol. 28, pp. 287–290, 1988.

[16] P.M.B. Vitányi, B. Awerbuch, *Atomic Shared Register Access by Asynchronous Hardware*, Proc. 27th IEEE Symposium on Foundations of Computer Science, pp. 233–243, 1986. (Errata, Ibid.,1987)