A Brief Survey of Concurrent Readers and Writers

Lefteris M. Kirousis*
Computer Technology Institute, University of Patras
P.O. Box 1122, 26110 Patras, Greece

Evangelos Kranakis
Centre for Mathematics and Computer Science
P.O. Box 4079, 1009 AB Amsterdam, The Netherlands

We are interested in implementations of concurrent high level objects from weaker lower level objects which are free from the usual control primitives, like mutual exclusion, test-and-set, etc. We give a survey of the most important recent results concerning such wait-free implementations of atomic multiwriter, multireader shared registers. We present several algorithms for constructing atomic bits from safe bits, atomic multivalued shared variables from atomic bits, as well as atomic multireader and multiwriter shared variables from multivalued single writer, single reader shared variables. We also discuss several methods for proving the correctness of concurrent reader concurrent writer protocols.

Qui scribit bis legit
Whoever writes reads twice
LATIN PROVERB

You read, but understood not;
for if you had understood,
you would have not condemned
C. P. CAVAFY

Research partially supported by the ESPRIT II Basic Research Actions Program of the EC ıder contract no. 3075 (Project ALCOM).

1. Introduction

The parallel execution of programs in multiprocessor environments and computer networks requires concurrency control of the various processes involved. If a read process and a write process share the same buffer area then precautions need to be taken so that the reader is protected from having its data garbled by the writer before completion of its action. By actively serializing concurrent actions using synchronization primitives like mutual exclusion, test-and-set, lockout, semaphores, etc., the data is effectively protected from such unexpected disturbances. Several schemes which guarantee exclusive access to a shared resource have been proposed both at the software and hardware level. However, an uncritical use of synchronization primitives in multiprocessor environments and parallel machines whose natural environment is meant to take advantage of the best aspects of parallelism would seem rather inappropriate. No doubt, there are several tasks, like access to a peripheral device, where exclusiveness is necessary, and therefore it is impossible to implement them without resort to the above synchronization primitives. However for other tasks (like airline reservation systems) exclusiveness is not so important. For example in the case of a shared file all that may matter is to be able to read the most recent correct version of the file.

Therefore the question arises as to which programming tasks can be implemented without resort to waiting mechanisms and how. Part of the answer seems to be that we need to discard Von Neumann's concept of centralized architecture and consider instead multiprocessor architectures with true parallelizable characteristics. Such multiprocessor environments are within the reach of today's technology [15]. However the main problem still remains. How can we coordinate the activities of the numerous processors at the software level in such a way that on the one hand we take full advantage of the merits of parallelism (e.g. increased speed of program executions) and on the other hand we guarantee the accurate implementation of the objects (like concurrent reading and concurrent writing) concerned? The answer to this question is not obvious. It is the purpose of the present paper to outline how to develop algorithms that implement 'objects' with optimal serializability characteristics via other lower level objects in a wait-free environment. Nevertheless, one thing seems clear. As has been pointed out by Lamport [24], to implement such primitives we need interprocess communication through a shared memory unit, also called shared register or shared variable.

It should also be noted that wait-free protocols are of particular interest not only because they are free from the usual control primitives (like, mutual exclusion, test-and-set, etc.) but also because they make possible a quantitative appraisal of the complexity of the various algorithms considered, e.g. determining the protocol with best running time.

2. The concurrent readers and writers problem

Throughout the present paper, the terms (shared) variable and (shared) register have identical meanings. In very simple terms the CRCW (Concurrent Readers, Concurrent Writers) problem can be described as follows. There is a

shared variable that is to be distributed and replicated among several processors. The distribution problem concerns the implementation of multivalued shared variables from boolean shared variables. The replication problem concerns reproducing the shared variable in order to be used by many processors. The required implementation is achieved by providing a protocol or algorithm that the concerned processors must follow. The main assumptions regarding the protocol constructions is that there should be

- no waiting (i.e. a processor should never have to wait for another processor to finish its action before it executes its own),
- completely asynchronous, with no synchronization primitives like mutual exclusion, semaphores, locking, test-and-set, etc,
- no global clock (i.e. the processors do not have access to a global time reference system).

The above assumptions define what the requirements for a proper solution should be. Moreover the only solutions to the CRCW problem we will be interested in are protocols that satisfy the above conditions.

We will not attempt to define here rigorously what a wait-free protocol is. Intuitively, however, by wait-free implementation of a concurrent-data object we understand a protocol or algorithm which can be executed by any processor in the system in a finite number of steps which is independent of the execution speeds of the processors involved. Several programs for implementing these protocols will be given later. Intuitively they should be constructed inductively using 'appropriate' initial assignment statements, and program statements constructed from these by iterating the two operators **if** $\cdots$ **then** $\cdots$ **else** $\cdots$, **for** $i := 1,...,n$ **do** $\cdots$ **od**, where $\cdots$ is a sequence of program statements at an earlier stage of the inductive construction [2].

2.1. The implementation problem

The implementation problem can be described as follows. We are given some registers each capable of storing words of a fixed length and with certain restrictions on their mode of operation, e.g. that only a certain number of processors are allowed to access each one of them; also, we assume that these registers satisfy certain serializability properties (i.e., a succession of reads and writes on the registers must, in some degree, be serializable). By using these registers (often called subregisters) as building blocks, we are asked to construct a more powerful compound register capable of storing words of (possibly) even longer length, with (possibly) more relaxed accessibility restrictions and the same or even stronger serializability properties. The subregisters will be structured according to an architecture (e.g., a matrix or a sequence of buffers) and moreover each operation (i.e., read or write) on the compound register will consist of a succession of operations on the subregisters executed according to a protocol. The architecture together with the protocol will guarantee the stronger properties we require from the compound register. The protocol will often utilize variables local to each processor. These local variables should, of course, be distinguished from the shared variables of the protocol which are none other than the subregisters used by the architecture.

In general, according to the strength of the serializability conditions they satisfy, we distinguish the following four types of registers: safe, normal, regular, and atomic. A register is called safe (respectively, normal, regular, atomic) if any system execution on the register (i.e., any succession of operation executions) is safe (respectively, normal, regular, atomic). For single writer registers, a system execution is safe if every read that does not overlap a write returns the most recent value, otherwise it may return an arbitrary value (with the restriction that it is within the set of values the register is allowed to assume). A system execution is normal if every read returns the value of a write that either precedes this read or is concurrent with it. A system execution is regular if it is normal and moreover no read returns a value that at the time this read begins has been already overwritten by another value.

Finally, a system execution is atomic if reads and writes behave as if they were linearly ordered. (This intuitive definition of atomicity is also valid for multiwriter registers, see Section 5.2.) More formally, a system execution is atomic if

1. There exists a total (i.e., linear) order extending the natural order in which the operation executions on the compound register have taken place (external consistency). This order, in some sense, represents the succession these operations seemingly follow.
2. There is no second write placed by this linear order between a read and the write it reads (internal consistency).

If we assume that there is a global time-reference system (i.e. that an observer could correctly time the beginning and end of all actions by all the subregisters) and if all subactions of an operation execution on the compound register precede in time all the subactions of a second operation execution on the compound register, then, by property (1) above, this order must place the second operation execution after the first one. In general, the natural order of the operation executions referred to at property (1) above, is imposed by the problem (e.g., it may be an acyclic relation that tells us if an operation execution can have an influence on another).

For simplicity we will use the following notations and abbreviations. By *mWnRbB* register we abbreviate an *m*-writer, *n*-reader, *b*-bit register. This means that the shared variable can hold values which are *b* bits long, can be written by *m* processors (the writers) and read by *n* processors (the readers). If we want to stress the fact that the register assumes the values $\{0,1,...,v-1\}$ (i.e. it is *v*-valued) then we write *mWnRvV*. Clearly, *mWnRbB* is equivalent to $mWnR2^bV$. A safe (normal, regular, atomic) $1W1R1B$ register is also called safe (normal, regular, atomic) bit. Safe bits are also known as 'flip-flops' (see Section 2.3) to indicate their resemblance to the bistable components used in computer manufacturing. Regular bits are also called 'flickering' bits to indicate the flickering behaviour of the shared variable over the duration of a write.

To facilitate our understanding the replication and distribution problems are conveniently split into the following easier constructions (however, this does not rule out the possibility of bypassing one or more of the steps below):

- flickering bits from flip-flops,
- atomic bits from flickering bits,
- atomic, single processor, multibit registers from atomic bits,
- atomic, single writer, multireader, multibit registers from atomic, single writer, single reader, multibit registers, and
- atomic, multiprocessor, multibit registers from atomic, single writer, multireader, multibit registers.

Thus the ultimate goal is the efficient construction of atomic, multiprocessor, multibit variables via flip-flops.

2.2. A brief history

The difficulty of implementing correctly concurrent transactions had already been recognized in the ALGOL68 report [49], where it is explicitely stated that 'in view of the none-to-advanced state of the art' only facilities 'restricted to their essentials' are introduced for 'collateral' (a weak form of concurrent) 'programming'. It was to overcome such problems that synchronization primitives like semaphores, mutual exclusion, test-and-set, etc., were introduced by [10] and [16] (see [6] for a very eloquent description of these primitives). The problem of concurrency control with readers and writers was also considered by [9], which offers two solutions to a restricted version of the CRCW problem. In their first solution they demand that 'no reader should be kept waiting unless a writer has already obtained permission to use the shared resource'. In the second solution they demand that not only 'the writers must have exclusive access while readers may share', but that in addition 'once a writer is ready to write it perfoms its write as soon as possible'. Clearly, neither of these solutions provide satisfactory answers to the CRCW problem posed here because they impose waiting on the processors.

The first to isolate and pose the problem of atomic, wait-free implementation of concurrent objects in its present 'single writer' form were L. Lamport [24] and G. Peterson [39]. Lamport discusses in [24] extensively the hierarchical nature of system executions, proposes protocols for distributing the value of a shared variable among two processors, one writing and the other reading, and classifies registers in three types according to increasing serializability characteristics: safe, regular and atomic. At the same time G. Peterson [39] considers the problem of distributing a shared variable among many readers, although at the time it was not known how to construct such shared variables from even simpler (non-atomic) ones. The multiwriter multireader problem remained dormant for a few years (and it almost looked as if it would turn out to be impossible to solve) till it was explicitely posed by N. Lynch in her MIT seminar series. Its most recent revival is due to P. Vitányi and B. Awerbuch [43] who attempted to implement wait-free multiprocessor protocols from the most basic shared variables possible, the 'idealized' flip-flops.

2.3. *Idealized flip-flops*

As mentioned above the basic problem in the CRCW area is how to implement atomic multiuser multivalued variables from the more fundamental single writer, single reader, single bit safe registers. But it must be stressed that such safe variables are only idealized objects of our imagination. The closest approximation of safe bits in the world of electronics is considered to be the flip-flop. These are one-bit memory elements that are capable of exhibiting either of two stable states (i.e. bistable). When triggered by sufficient current they can switch from voltage 0 to voltage 1 and vice versa. However when the input to the flip-flop is marginal it may cause the bistable device to remain for an indefinite period of time in a metastable state which is different from any of the above two states. Mathematically, this phenomenon is due to the fact that a continuous function assuming the values 0 and 1 will also have to assume any value between 0 and 1 (known in mathematical analysis as Bolzano's theorem). For a formal description of this phenomenon we refer the reader to [30]. However we will not be concerned here with such situations. The idealized safe bits we have in mind are 'perfect' bistable devices that do not exhibit the above metastability phenomenon.

3. Initial algorithms

Here we study several relatively simple implementations of the CRCW problem. Although we will never mention it explicitely we assume that all the registers given below begin the execution of their respective protocols with some consistent initialization. In order to avoid unnecessary notational complications we describe all our protocols rather informally using the basic statements **write** and **read**. It should be stressed however that these are merely assignment statements of the form $x := F(y_1, \ldots, y_n)$, where F is a function symbol in the protocol language and $x, y_1, \ldots, y_n$ are protocol variables.

3.1. *Two examples*

One of the most surprising aspects of the CRCW area is the ease with which one can provide either unsatisfactory or wrong algorithms. We illustrate this in the present section with two examples. In both instances we want to implement a $1W1RbB$ atomic variable using atomic bits. We represent a given value $0 \leqslant v < 2^b$ by its binary encoding, with b bits. The main idea in executing read and write actions in the compound register is the following. To write a value v the writer writes the binary encoding of v in a track (or buffer) consisting of b spaces (subregisters), one for each bit. To read a value the reader enters a certain track (specified by the algorithm), reads all the bits in the track and returns the sequence of b bits it read.

In our first example, there is an infinite sequence of tracks $tr_1, tr_2, \ldots$ each consisting of b atomic bits that can be read by the reader and written by the writer. Next we consider the following algorithm [13]; x is an atomic shared variable that can be written by the writer and read by the reader. To write a value v the writer executes the following protocol.

write value v on track tr_{x+1};

$x := x + 1$;

To read a value the reader executes the following protocol.

read x;

return value from tr_x;

In the above protocol, the reader always reads the most recently written value and this value is always correct. There should be no difficulty in verifying that this register is atomic. However there is a slight problem with this register. It uses infinite space, namely an infinite number of length b tracks.

Therefore one is lead to limiting the number of tracks. In our second example we assume that there are only three tracks, numbered tr_1, tr_2, tr_3 [25]. Consider a function f such that $f(x,y) \neq x,y$, for all $x,y = 1,2,3$. Further let K_R, K_W be two $1W1R$ three-valued atomic registers. K_R (respectively, K_W) is written by the reader (respectively the writer) and read by the writer (respectively the reader). To write a value v the writer executes the following protocol.

read K_R;

write v on track $tr_{f(K_R,K_W)}$;

write new track number $f(K_R, K_W)$ on K_W;

To read a value the reader executes the following protocol.

read K_W;

write track value read above in K_R;

return the value on track numbered tr_{K_W};

This register satisfies the necessary bounded space requirements. It uses three tracks each of length b. Hence the total space used is $3 \cdot b + O(1)$ and the time is $b + O(1)$. Unfortunately it fails to be atomic. A moment's reflection will reveal that the reader and the writer may very well collide on the same track, in which case the reader could report a 'meaningless' value. We will see in Section 4.1 how to resolve this subtlety.

3.2. The basic implementations

We now begin by considering protocols that 'gradually' solve instances of the CRCW problem. The first simple construction concerns implementing regular bits with safe bits and is due to L. Lamport [24]. Suppose that K represents a safe bit. We implement a regular bit K' as follows. The reading operation is performed by merely reading K. Concerning the writing operation, the main 'trick' is that in order to write the value b the writer 'avoids writing the same value twice', i.e. let l be a variable which is local to the writer.

if $l \neq b$ **then**

write b on K;

$l := b$;

fi

We can prove the following simple theorem.

THEOREM 3.1. ([24]). *A regular bit can be implemented with a safe bit.*

Next we proceed with multivalued registers. Here we are concerned with the distribution problem. We implement a single writer b-bit register by using only single writer single bit registers. As opposed to the previous examples where we used binary encoding, we will now use unary encoding, i.e. the value v is represented by $v-1$ zeroes followed by a 1 at the v-th position. Suppose that $K_1, \ldots, K_n$ are single bit, single writer registers and let K be the n-valued register in which the write and read operations are performed as follows. To write a value v the processor writes the bit 1 on register K_v and sets all the previous registers $K_1, \ldots, K_{v-1}$ to 0 beginning with K_{v-1} and moving backwards all the way down to K_1. More formally, in order to write the value v the processor executes the following protocol.

write 1 on K_v;

for $u := v-1$ **step** -1 **until** 1 **do write** 0 on K_u **od**;

To read a value the processor returns the first v such that $K_v := 1$ but for all $u < v$ $K_u := 0$. More formally, in order to read a value the processor executes the following protocol.

$v := 1$;

while K_v has value 0 **do** $v := v+1$ **od**;

return v

This protocol constitutes construction 4 in Lamport's paper [24] which also proves the following theorem.

THEOREM 3.2. ([24]). *A regular, single writer, m-reader, n-valued register can be implemented with n regular, single writer, m-reader, single bit registers.*

It is not hard to see that K can fail to be atomic even if all the K_v's are atomic. For example, consider the initial value $\mathbf{3} = (0,0,1)$ in the above implementation of a 3-valued register with three atomic subregisters. In the following sequence of actions the second read $read_2$ returns the old value **2** while the first read $read_1$ returns the new value **1**.

read$_1$: reads $x_1 = 0$ and continues
write$_1$ (writes value **1**): writes $x_1 = 1$ and ends
write$_2$ (writes value **2**): writes $x_2 = 1$ and continues
read$_1$: reads $x_2 = 1$ and returns **2**
read$_2$: reads $x_1 = 1$ and returns **1**
write$_2$: writes $x_1 = 0$ and ends

However in a rather clever twist it was observed by K. Vidyasankar [45] that a simple modification of the read operation can lead to an atomic register. All that the new modified reader algorithm needs to do is after finding the first 1, i.e. the first v with $K_v = 1$, not to report v as the value read, but instead to backstep, read the values K_{v-1} through K_1 over again and report instead the smallest $u \leqslant v$ such that $K_u = 1$. The writer protocol remains unchanged. To read a value in the modified register K' the processor executes the following protocol.

$v := 1$;
while K_v has value 0 **do** $v := v + 1$ **od**;
$u := v$;
for $i := v - 1$ **step** -1 **until** 1 **do if** K_i has value 1 **then** $u := i$ **od**;
return u;

For this new modified register we can prove that K' is atomic if each K_v is. In fact we have the following theorem.

THEOREM 3.3. ([45]). *An atomic, single writer, m-reader, n-valued register can be implemented with n-atomic, single writer, m-reader, single bit registers.*

There are several problems with this last algorithm. For once it does not replicate the shared variable among many users. It would also be wrong to think that Theorem 3.3 has solved the distribution problem of implementing an atomic multivalued variable from atomic bits. A moment's reflection will show the disadvantages of using the unary encoding of numbers. The above algorithm implements a single writer, b-bit, atomic variable by using an exponential number (i.e. 2^b) of single writer, atomic bits. We will show in Section 4.1 that this can be done more efficiently by using only $O(b)$ atomic bits, in the $1W1RbB$ case, and in Section 4.2 in $O(n^2 \cdot b)$ atomic bits, in the $1WnRbB$ case. Finally, the most general $nWnRbB$ case is studied in Section 4.3. It is exactly this interplay between time and space complexity of an algorithm and the difficulty of proving its correctness that creates numerous interesting and unexpected complications in this area.

Here is a summary of the main single writer implementations given in the present section. The first column is the paper, and the second column gives the number and type of subregisters required to implement the register in the third column.

PAPER	SUBREGISTER	REGISTER
[24]	1 Safe 1*WnR* 1*B*	Regular 1*WnR* 1*B*
[24]	2^b Regular 1*WnR* 1*B*	Regular 1*WnRbB*
[45]	2^b Atomic 1*WnR* 1*B*	Atomic 1*WnRbB*

FIGURE 1. Implementing stronger registers from weaker ones

4. THE MAIN ALGORITHMS

We can now proceed with some of the main solutions to the CRCW problem. Here we show how to implement atomic multiwriter, multireader, multivalued shared variables using as building blocks the idealized flip-flops (also called safe bits).

4.1. Atomic bits

There are two important points that must be considered in implementing 1*W*1*RbB* registers. The first is that in order to minimize the space used we must employ binary encoding of the values concerned. This means that we will write the values on a finite number of tracks (or buffers) each consisting of b single bit subregisters. Clearly on such a b-bit buffer we will be able to write and read values in the range from 0 to 2^b-1. The second is that we will implement a switch (i.e. protocol together with some extra bits) that will eventually enable the processors to alternate among the tracks in order to read and write, respectively, values in such a way that the desired correctness properties of the protocol are satisfied. In doing this we are lead to considering two types of protocol implementations. In the first one pure copies are used, i.e. the copies of the simulated object are never overwritten while they are being used, while in the second one they may not necessarily be pure. There is an interesting interplay between these two types of constructions. As expected, the pure copies construction uses more space but less time, while the reverse is true in the impure copy construction.

Using impure copies G. Peterson [39] implements an atomic 1*WnRbB* register with atomic 1*WnR* bits. Peterson's idea is roughly as follows. The desired shared variable is implemented on three tracks. The writer writes the value three times, once on each track, and the reader reads the values from each of the tracks but in different order than the writer. He then implements a switch that permits the reader to decide which value was read 'without collision'. The switch must consist of atomic bits since it must able to detect collisions in the three tracks. Next L. Lamport [24] implements an atomic bit using a finite number of safe bits. Combining these two implementations we obtain a modular construction of an atomic, single writer, single reader register.

THEOREM 4.1. ([24,39]). *Using impure copies, an atomic, single writer, single reader, b-bit register can be implemented, with* $3\cdot b+O(1)$ *safe, single writer, single reader bits, in time* $3\cdot b+O(1)$.

In the sequel we present a modified version of Peterson's construction due to [47]; the new protocol has a simpler control structure and both the read and write operations do less work. The atomic $1W1RbB$ register we present consists of three atomic bits: *c*, *writing*, *reading*, three safe *b*-bit tracks: tr_0, tr_1 and the copy track *ctr*. The writer also uses two local boolean variables *cl*, *wl*. ($\oplus$ denotes modulo 2 addition.) To read a *b*-bit value the processor executes the following protocol.

read bit b_1 from *writing*;
write bit $b_1 \oplus 1$ to *reading*;
read bit *b* from *c*;
read track tr_b;
read bit b_2 from *writing*;
if $b_2 = b_1$ **then**
    return value from tr_b
else
    return value from *ctr*
fi

To write a *b*-bit value *v* the processor executes the following protocol.

$cl := cl \oplus 1$;
write value on tr_{cl};
write *cl* in *c*;
read bit b_1 from *reading*;
if $b_1 \neq wl$ **then**
    write value *v* on *ctr*;
    write b_1 on *writing*;
    $wl := b_1$
fi

Combining this algorithm with the algorithm in [24] we obtain the following improvement of Theorem 4.1.

THEOREM 4.2. ([24,47]). *Using impure copies, an atomic, single writer, single reader, b-bit register can be implemented, with* $3 \cdot b + O(1)$ *safe, single writer, single reader bits, in time* $2 \cdot b + O(1)$.

Another idea is to implement a switch that will enable the processors to alternate on the tracks but in such a way that they never have to execute a write or read action on the same track at the same time. Since in this case the tracks are collision free they can be assumed to consist of safe bits. Also, unlike Peterson's original construction, the switch of the four track register need only be regular, which gives a direct implementation of the desired register from regular subregisters. Thus, using pure copies [19] implements an atomic $1W1RbB$ register with $4 \cdot b + 39$ safe $1W1R$ bits, in time $b + 26$; this was later

improved by [42] to $4 \cdot b + 8$ safe $1W1R$ bits, in time $b + 4$. Hence we obtain a direct construction of the required register in the following theorem.

THEOREM 4.3. ([19,42]). *Using only pure copies, an atomic, single writer, single reader, b-bit register can be implemented, with* $4 \cdot b + O(1)$ *safe, single writer, single reader bits, in time* $b + O(1)$.

Next we present a slightly weaker version of Tromp's four track register (for details on his $4 \cdot b + 8$ construction consult [42]). The register consists of the following subregisters. Four safe b-bit tracks divided into two groups: $T_a = \{tr_{a,0}, tr_{a,1}\}$, $a = 0, 1$. An atomic bit W (R) to be written by the writer (reader) and read by the reader (writer) and two atomic 'display' bits D_0, D_1 to be written by the writer and read by the reader. Also the writer has the local variables w, d_0, d_1 and the reader the local variable r. To write a b-bit value the writer executes the following protocol.

if $R = w$ **then**
  $w := w \oplus 1$;
  write value on track tr_{w,d_w};
  $W := w$;
else
  $d_w := d_w \oplus 1$;
  write value on track tr_{w,d_w};
  $D_w := d_w$
fi

To read a b-bit value the reader executes the following protocol.

if $W \neq r$ **then**
  $r := r \oplus 1$;
  $R := r$
fi;
read bit a from D_r;
read b-bit value from track $tr_{r,a}$

Next we summarize the space and time complexity of the above constructions. First, some explanations are needed for the proper interpretation of the complexity tables given in the sequel. By time complexity of a high-level object we understand the worst-case number of accesses of low-level subregisters required by the given processors, which for the CRCW problem are readers and writers, in order to complete a full run of their protocol. By space complexity of a high-level object we understand the number of low-level subregisters used for the implementation of the object multiplied by the bit-size of the tags (or time stamps) used by each individual subregister. With this in mind we have the following table.

PAPER	SPACE	TIME
[39] + [24]	$3 \cdot b + O(1)$	$3 \cdot b + O(1)$
[47] + [24]	$3 \cdot b + O(1)$	$2 \cdot b + O(1)$
[19]	$4 \cdot b + 39$	$b + 26$
[42]	$4 \cdot b + 8$	$b + 4$

FIGURE 2. From $1W1R$ safe bits to $1W1RbB$ atomic registers

Concerning the optimality of the atomic bit implementations we can prove the following theorem.

THEOREM 4.4. ([38,42]). *3 safe bits, 2 written (read) by the writer (reader) and one by the reader (writer) are necessary and sufficient to implement an atomic bit.*

4.2. Atomic multireader registers

We now come to a more complicated problem. It concerns the construction of multireader shared variables which are written by a single writer, from single writer, single reader registers. This problem was solved simultaneously by four different groups of researchers L. Kirousis, E. Kranakis and P. Vitányi [19], J. Anderson, A. Singh and M. Gouda [5], G. Peterson and J. Burns [36], R. Newman-Wolfe [33] by using entirely different algorithms. A new algorithm was later reported by A. Israeli, M. Li and P. Vitányi [18]. Two among these five papers, [19] and [33], implement algorithms that use pure copies. As such they achieve better reader-time. Here is a table of the known implementations together with their complexity.

PAPER	SPACE	WRITER-TIME	READER-TIME
[19]	$O(n^2 \cdot (n+b))$	$O(n \cdot b)$	$O(n+b)$
[5]	$O(n \cdot (n+b))$	$O(n \cdot (n+b))$	$O(n \cdot b)$
[36]	$O(n^2 \cdot b)$	$O(n \cdot b)$	$O(n \cdot b)$
[33]	$O(n \cdot (n+b))$	$O(n \cdot b)$	$O(n+b)$
[18]	$O(n^2 \cdot b)$	$O(n \cdot b)$	$O(n \cdot b)$
[28]	$O(n^2 \cdot b)$	$O(n \cdot b)$	$O(n \cdot b)$

FIGURE 3. From $1W1R1B$ atomic to $1WnRbB$ atomic

In addition [46,48] reports two simple constructions of atomic, multireader, multivalued registers from multireader, atomic bits.

4.3. Atomic multiwriter registers

Finally we come to the most difficult problem in this area. Implementing a multiwriter shared variable.

The first such multiwriter implementation due to B. Bloom was rather unique for its simplicity, but it concerns only the construction of a two writer atomic register. Suppose that K_0, K_1 are two $b+1$-bit, atomic, single writer

registers such that K_0 (K_1) can be written by the writer (reader) and written (read) by the reader (writer). Let $p_0, p_1, \ldots, p_{n-1}$ be n processors (all readers) of which the first two p_0, p_1 can both read and write. We represent the contents of the registers as pairs (t_i, v_i), where $t_i \in \{0,1\}$ is a tag and v_i is the actual b-bit value. Also let $\oplus$ denote modulo 2 addition. The algorithm for implementing the two writer register is as follows. Writer p_i, $i=0,1$, writes a value by first reading the tag of $K_{i\oplus 1}$ and then writing its value on K_i together with an appropriate tag.

p_i WRITES THE VALUE v:
read the tag $t_{i\oplus 1}$ from $K_{i\oplus 1}$;
write $(i \oplus t_{i\oplus 1}, v)$ to register K_i;

The reader p_i ($i=0, \ldots, n-1$) reads twice. At first, it reads the tags in both K_0, K_1, say t_0, t_1 and then returns the value it reads in register $K_{t_0 \oplus t_1}$, after a second reading.

p_i READS A VALUE:
read the tag t_0 from K_0;
read the tag t_1 from K_1;
return the value from $K_{t_0 \oplus t_1}$;

(Incidentally, notice that all the readers execute exactly the same protocol.) It can be shown that the resulting two writer register is atomic [8]. Hence we have the following theorem.

THEOREM 4.5. ([8]). *An atomic $2WnRbB$ register can be implemented with two atomic $1WnR(b+1)B$ registers.*

The first attempt at solving the general multiwriter problem with bounded tags was by P. Vitányi and B. Awerbuch [43]. This construction uses single writer, single reader registers to implement the desired register, but the 'base' registers must have unbounded tags. Here is a description of the algorithm. The $n \times n$ matrix register K consists of n^2 atomic, single writer, single reader, subregisters $K_{i,j}$, $i,j=1,\ldots,n$. Each processor p_i is connected to the write terminal of $K_{i,j}$ and the read terminal of $K_{j,i}$, $j=1,\ldots,n$. Each subregister can hold a pair (*tag*, *value*); tags are pairs (k,i), where k is an arbitrary non-negative integer and $i=1,\ldots,n$, while *value* is the actual value which will be either read or written. The writer reads its column, updates its tag and then writes its updated tag and value to all subregisters in its row.

p_i WRITES THE VALUE v:
for $j:=1,\ldots,n$ **read** $K_{j,i}$;
compute the lexicographically largest tag $(k_{\max}, m)$ among the tags just read and **set** own tag to $(k_{\max}+1, i)$;

for $j := 1,..., n$ **write** on $K_{i,j}$ the pair $((k_{\max}+1, i), v)$;

The reader reads its column, stores the value corresponding to the maximal tag and updates its tag. It then writes its updated tag as well as the value it stored to all subregisters in its row.

p_i *READS A VALUE*:
for $j := 1,..., n$ **read** $K_{j,i}$;
compute the lexicographically largest tag $(k_{\max}, m)$ among the tags just read, store the value v_m included in this pair and **set** own tag to $(k_{\max}, m)$;
for $j := 1,..., n$ **write** on $K_{i,j}$ the pair $((k_{\max}, m), v_m)$;
return v_m;

Inspecting the above protocol it is not difficult to see that the 'diagonal' registers $K_{i,i}$ are not shared, but are only used by processor p_i, for $i = 1,..., n$, respectively; hence they are not necessary. For this multiwriter, multireader register we can prove the following theorem.

THEOREM 4.6. ([4,43]). *The matrix register is an atomic, n-writer, n-reader register which can be implemented with* $n^2 - n$ *atomic, single writer, single reader registers with unbounded tags.*

Despite its 'elegance', the space complexity of the matrix register is infinite and as such this register does not provide a solution to the CRCW problem. A subsequent implementation, due to G. Peterson and J. Burns [37], used some of the ideas of the bounded tag version of the matrix algorithm [43] (which was later found to be erroneous) together with the new idea of repeated reading in order to restrict the number of time stamps to a finite number, but their algorithm was later found to be wrong by R. Schaffer [41]. This last paper corrects the [37]-register and provides a correctness proof of the new implementation using input output automata. Another construction was also given by M. Li and P. Vitányi [29]; its correctness proof is given in [28]. Here is a table of the known multiwriter, multireader, atomic registers together with their complexity.

PAPER	SPACE	TIME
[43]	∞	$O(n)$
[37] + [41]	$O(n^3 \cdot b)$	$O(n^3 \cdot b)$
[17]	$O(n^4 \cdot b)$	$O(n^2 \cdot b)$
[29] + [28]	$O(n^3 \cdot b)$	$O(n \cdot b)$

FIGURE 4. From $1W1R1B$ atomic to $nWnRbB$ atomic

4.4. *Interfaces for multiprocessor registers*

An interesting phenomenon appearing in the construction of atomic registers is the simplicity of the protocols concerned if we could assume an infinite tag system (i.e. infinite space). This phenomenon manifests itself in the case of the matrix register which is relatively easy to prove atomic if the base registers were assumed to possess unbounded tags. However the same problem is surprisingly hard in the bounded tag case. One is therefore led to the following interesting question. Can we construct interfaces for converting a given unbounded tag algorithm (like the matrix register) into a bounded tag algorithm? A. Israeli and M. Li [17] were the first to consider this problem; in this paper they construct bounded sequential time stamp systems in the case where no two operations are concurrent. A significant improvement is reported by D. Dolev and N. Shavit in [11] which provides bounded concurrent time-stamp systems.

5. Proving correctness

Proving the correctness of CRCW protocols can always create heated debates among researchers in the field. Time and again the usual intuitive approach for proving correctness consisted in proving first the correctness for unbounded tag registers, and then showing that in fact a finite number of tags is sufficient. The first part has usually been easy to do; after all you design the algorithm with this property in mind. The second part is usually the main point of contention. Usually, you need to make some adjustments and modifications to the initial infinite tag protocol in order to convert it into a finite tag protocol. But alas this often complicates the correctness proof of the original protocol to such an extend that you run the risk of losing track of what you are trying to do. Such examples abound in this research realm and must have certainly been experienced by any researcher. We hope that the reader has been convinced of that by the two examples given at the beginning of Section 3. Therefore correctness proofs are not something to be viewed lightly in this domain, but on the contrary it is an area of vital importance.

We have not given the proofs of any of the theorems in the paper. However it should be stressed that most of them (especially the ones on multireader and multiwriter protocols) require rather laborious proof techniques (the single exception being the implementation of a flickering bit from a flip-flop). In the sequel we consider several proof techniques that have been used to prove the correctness of the CRCW protocols and refer the interested reader to the original papers.

5.1. *Lamport's semantics*

L. Lamport [24] introduces register axioms and system executions for single writer registers in order to give rigorous proofs of the atomicity of his implementation of an atomic single writer, single reader register from safe bits. The main notion providing the ability to build hierarchicaly the compound registers is that of system execution. These are triples $\mathcal{S} = (A, \longrightarrow, -\rightarrow)$. Intuitively, $\longrightarrow$ stands for 'precedes' while $-\rightarrow$ stands for 'can causally affect'. A is a

countable set representing the set of read and write actions of an execution, and $\longrightarrow, -\!\!\rightarrow$ are binary relations satisfying axioms A_1, A_2, A_3, A_4, A_5 below, for all $a,b,c,d \in A$:

A_1: $\longrightarrow$ is an irreflexive, partial ordering,
A_2: if $a \longrightarrow b$ then $a -\!\!\rightarrow b$ and not $(b -\!\!\rightarrow a)$,
A_3: if $a \longrightarrow b -\!\!\rightarrow c$ or $a -\!\!\rightarrow b \longrightarrow c$ then $a -\!\!\rightarrow c$,
A_4: if $a \longrightarrow b -\!\!\rightarrow c \longrightarrow d$ then $a \longrightarrow d$,
A_4^*: if $a -\!\!\rightarrow b \longrightarrow c -\!\!\rightarrow d$ then $a -\!\!\rightarrow d$,
A_5: the set $\{b \in A : \text{not}\,(a -\!\!\rightarrow b)\}$ is finite,
A_6: $a \longrightarrow b$ or $b -\!\!\rightarrow a$.

If in addition the system execution satisfies axiom A_6 then it is called a global system execution. Global system executions are intended to capture the natural order of actions when there is a global time reference system (of which the processors need not be aware). By global time models we understand triples (A,s,f), where $s,f : A \mapsto \mathbb{R}$ are real-valued functions such that for all a, $s(a) \leqslant f(a)$ and for all real numbers t, the set $\{a \in A : s(a) < t\}$ is finite. Every global time model gives rise to a global system execution. Simply define

$$a \longrightarrow b \Leftrightarrow f(a) < s(b), \text{ and}$$

$$a -\!\!\rightarrow b \Leftrightarrow s(a) \leqslant f(b).$$

But also conversely, we can prove the following theorem.

THEOREM 5.1. ([23]). *Every global system execution is isomorphic to a global time model. More generally, every system execution can be extended to a global system execution.*

Another class of system executions is obtained from the set A of all nonempty subsets of a partially ordered set $(P, <)$. The system execution defined by

$$a \longrightarrow b \Leftrightarrow \forall x \in a \forall y \in b (x < y), \text{ and}$$

$$a -\!\!\rightarrow b \Leftrightarrow \exists x \in a \exists y \in b (x < y),$$

satisfies axioms $A_1, A_2, A_3, A_4, A_4^*$. (According to [7] this last axiom was first proposed by Abraham in [1].) Extending Theorem 5.1, S. Ben-David [7] (and independently Anger) proves a completeness result for Lamport's axiomatic system.

THEOREM 5.2. ([7]). *For every global system execution $\mathcal{S} = (A, \longrightarrow, -\!\!\rightarrow)$ on a set A of actions satisfying axiom A_4^* there is a class $\mathcal{G} = \{g = (A, \longrightarrow_g, -\!\!\rightarrow_g)\}$ of global time models with the same set A of actions such that for all $a,b \in A$,*

$a \longrightarrow b$ *if and only if* $\forall g \in \mathcal{G}(a \longrightarrow_g b)$, *and*
$a -\!\!\rightarrow b$ *if and only if* $\forall g \in \mathcal{G}(a -\!\!\rightarrow_g b)$.

It follows that global time models capture the full provability power of global system executions satisfying axiom A_4^*. [24] also gives three communication axioms B_0, B_1, B_2 concerning operation executions on the same single writer register. Axiom B_0 states that the write actions on the same register must be linearly ordered by $\longrightarrow$, say

$$w^0 \longrightarrow w^1 \longrightarrow \cdots \longrightarrow w^k \longrightarrow \cdots$$

Axiom B_1 states that for any read r and any write w to the same register either $w - \rightarrow r$ or $r - \rightarrow w$, while B_2 states that a read obtains one of the values that may be written in the register. A read r is said to see $w^{[i,j]}$ where $i = \max\{k : \text{not}(r - \rightarrow w^k)\}$ and $j = \max\{k : w^k - \rightarrow r\}$. Next Lamport defines three types of single writer shared variables.

- **safe**: a read that sees $w^{[i,i]}$ obtains the value w^i,
- **regular**: a read that sees $w^{[i,j]}$ obtains the value w^k, for some $i \leqslant k \leqslant j$,
- **atomic**: if a read sees $w^{[i,j]}$ then $i = j$.

Notice that the above definition of atomicity refers to registers that are *actually* atomic, i.e., no overlapping of a read with more than one write is allowed (overlapping of reads that read the same write is allowed, but this obviously does not affect the serializability of the operation executions). In other words, operation executions are essentially *a priori* linearly ordered. In contrast to this approach, in Section 2.1 we defined atomicity in a virtual sense, i.e. we allowed overlappings of a read with many writes, but we put as a requirement the existence of a linear order which represents the succession that operation executions seemingly follow. The two notions are connected in the following theorem:

THEOREM 5.3. ([24]). *Let* $\mathcal{S} = (A, \longrightarrow, - \rightarrow)$ *be a system execution on a regular register* w *such that there exists an integer-valued function* ϕ *on the set of reads satisfying*

1. *if* r *sees* $w^{[i,j]}$ *then* $i \leqslant \phi(r) \leqslant j$,
2. r *returns the value* $w^{[\phi(r)]}$,
3. *if* $r \longrightarrow r'$ *then* $\phi(r) \leqslant \phi(r')$,

for all reads r, r'. *Then* $\mathcal{S}$ *implements a system execution in which* w *is an atomic register.*

This proof technique has been used in [24] to prove the correctness of his algorithms in Subsections 3.2, 4.1.

5.2. Semantics with a single causality relation

This semantics was developed by B. Awerbuch, L. Kirousis, E. Kranakis and P. Vitányi in [4] in order to prove the correctness of atomic, multiwriter registers. In a general readers/writers protocol there are two types of actions which are being executed: reads and writes. Let A be the set of such actions associated with an execution of the protocol. Let R and W be the sets of read and write actions, respectively, such that $A = R \cup W$ and $R \cap W = \varnothing$. It is possible

to provide a proof technique based on a single binary relation (partial ordering) $\rightarrow$ instead of the two $\longrightarrow$, $-\rightarrow$ of Lamport's semantics. Intuitively, $a \rightarrow b$ stands for 'a precedes b but b does not causally affect a'. In this semantics, a is said to be concurrent with b if neither $a \rightarrow b$ nor $b \rightarrow a$. Also, a write w is said to directly precede a read r if $w \rightarrow r$ and there is no write w' such that $w \rightarrow w' \rightarrow r$. In addition to the precedence relation $\rightarrow$, it is assumed that there is a reading function $\pi: R \mapsto W$ associating with each read action $r \in R$ the write action $\pi(r) \in W$ which is read by r. The triple $\rho = (A, \rightarrow, \pi)$ is also called a run of the protocol. We distinguish the following types of runs $\rho = (A, \rightarrow, \pi)$.

- **normal**: $\pi(r)$ either precedes r or is concurrent with r.
- **regular**: $\pi(r)$ either directly precedes r or is concurrent with r.
- **atomic**: There is a total order $\Rightarrow$ extending $\rightarrow$ (external consistency), there is no write w such that $\pi(r) \Rightarrow w \Rightarrow r$ (internal consistency) and $\pi(r) \Rightarrow r$.

(The definition of normal register is due to L. Meertens [31].)

The clan $[w]_\rho$ of a write w is the set $\{w\} \cup \{r \in R: \pi(r) = w\}$. Define the relation $w \rightarrow^\rho w'$ to mean that for some $a \in [w]_\rho$, $a' \in [w']_\rho$, $a \rightarrow a'$. In proving the atomicity of registers the following criterion has proved to be useful.

THEOREM 5.4. ([4]). *For any run ρ of a register,*

1. *ρ is atomic $\Leftrightarrow$ ρ is normal and $\rightarrow^\rho$ is acyclic.*
2. *For single writer registers, ρ is atomic $\Leftrightarrow$ ρ is regular and π is weakly monotonic.*

The above proof technique has been further elaborated in [4] to include communication axioms for multiwriter registers and was used successfully to prove the correctness of the matrix register [43], the two writer register [8], the four track registers [19] and [42] and the multiwriter register in [28,29].

Clearly, the theorem is an extension of Theorem 5.3 to multiwriter registers. The first part of the theorem had been discovered independently by K. Vidyasankar [44] in the context of database serializability [34,35]. An equivalent axiomatic formulation of the theorem was given by G. Peterson and J. Burns [36]. They provide a list of seven axioms $BC_1 - BC_7$ which must be satisfied by a precedence relation in order to be atomic. They also point out that for the case of single writer registers these seven axioms reduce to the last two BC_6, BC_7. However, their axiomatization does not in any way enhance or shorten the atomicity proofs.

In some cases it may be possible to associate an open time interval $(s(a), f(a))$ to each action $a \in A$, where $s, f: A \mapsto \mathbf{R}$ are real-valued functions; $s(a)$ is the starting and $f(a)$ the finishing time of the action. An important aspect of atomic runs concerns the instantaneousness of their execution, i.e. although their actions have an actual duration (possibly overlapping one another) in a global time reference system each individual action may be considered to take place at a particular instant. If the causality relation $\rightarrow$ is induced by a representation of the actions as open time intervals then it is possible to 'shrink' the duration of the actions to a time instant. A shrinking

function is a one to one mapping associating to each interval $(s(a), f(a))$ a real number $\sigma(a)$ in this interval. The relation $a <^{\sigma} b$ defined by $\sigma(a) < \sigma(b)$ induces an ordering on the set of actions. A run ρ is called shrinking atomic if there is a shrinking function such that $(A, <^{\sigma}, \pi)$ is atomic. More formally we can prove the following theorem.

THEOREM 5.5. ([4]). *For any run ρ of a register, if $\rightarrow$ is the order induced by the representation of the actions A by open time intervals then $\rho = (A, \rightarrow, \pi)$ is atomic if and only if ρ is shrinking atomic.*

An equivalent definition of atomicity using shrinking of the actions was also given by J. Misra [32].

5.3. Input/Output automata

The Input/Output automaton provides a very powerful methodology for modelling and proving the correctness of concurrent and distributed discrete event systems. It has been defined by N. Lynch and M. Tuttle (see [27] for an outline of the model). This model has been used in [26] to prove the correctness of the Bloom register [8], the matrix register [43], [4], as well as by [41] to correct the modified multiwriter algorithm in [37].

It is also interesting to note that J. Tromp [42] in the full version of his paper develops a finite state automaton, the atomicity automaton, for proving the correctness of his four track protocol. Transitions in the atomicity automaton represent beginnings and ends of read and write actions, while nodes represent the atomicity state of the shared variable. This makes possible the application of automatic verification methods for proving the correctness of the protocol (this is of practical interest as well, and ought to satisfy even the most skeptical).

6. CONCLUSION

Due to both hardware and software design-constraints conventional programming methodology has limited itself mainly to environments with serial mode of operation. However, this is not only too restrictive, but in addition, runs contrary to the parallelism encountered by many physical machines of everyday life [12]. Although we are still far from creating a thorough framework for writing, implementing and proving the correctness of parallel programs, there is no doubt that there are several programming tasks which are naturally amenable to parallel methodology. Surprisingly, we have shown that the CRCW problem is such a parallelizable task.

The solutions of the CRCW problem presented here also raise a very relevant question. Can we implement higher level register objects (like atomic test-and-set, mutual exclusion, etc.) using atomic registers? It was shown by M. Herlihy [14] and M. Loui and H. Abu-Amara [22] that this is indeed impossible. For more information consult [2] and [20]. The impossibility results just cited have also given rise to another line of research. It concerns randomized, wait-free implementations for the above primitives. For more details consult

[21], [3], [40].

In the present paper we have outlined some of the most important algorithms and proof techniques in the concurrent readers and writers area. The simplicity of the problem as well as its importance for implementations in parallel environments provoked a flurry of activity by numerous researchers which lifted the problem from its long dormancy. Although concurrent reading and concurrent writing is now better understood and its wait-free feasibility is beyond doubt, correctness proofs of the existing protocols (especially the multiwriter ones) are still complicated and hard to comprehend. It is therefore important that more efforts are directed at

- providing new, simpler algorithms for implementing the concurrent objects concerned,
- refining and amplifying existing proof methods using order semantics, that will clarify and illuminate our understanding of parallelizable programming methodology.

7. Acknowledgements

We are thankful to P. Vitányi for numerous discussions on the topics presented in this brief survey. John Tromp provided useful comments on a first draft of the paper.

References

1. U. Abraham, S. Ben-David (1987). Informal and formal correctness proofs for programs (for the cirtical section problem. Reprint.
2. J.H. Anderson, M.G. Gouda (1987). *The Virtue of Patience: Concurrent Programming with and without Waiting,* Technical Report 78712-1188, Department of Computer Science, University of Texas.
3. James Aspnes, Maurice Herlihy (1988). *Fast Randomized Consensus using Shared Memory,* Technical Report CMU-CS-88205, Carnegie Mellon.
4. Baruch Awerbuch, Lefteris M. Kirousis, Evangelos Kranakis, Paul Vitányi (1988). On proving register atomicity. K. Nori, S. Kumar (eds.). *Proceedings of the 8th Conference on Foundations of Software Technology and Theoretical Computer Science,* Springer Verlag Lecture Notes in Computer Science, Heidelberg, Vol. 338.
5. J.H. Anderson, A. Singh, M.G. Gouda (1987). The elusive atomic register. *Proceedings of 6th ACM Symposium on Principles of Distributed Computing, Vancouver, Canada.*
6. M. Ben-Ari (1982). *Principles of Concurrent Programming,* Prentice Hall International.
7. S. Ben-David (1988). The global time assumption and semantics for concurrent systems. *Proceedings of 7th ACM Symposium on Principles of Distributed Computing, Toronto, Canada.*

8. Bart Bloom (1987). Constructing two-writer atomic registers. *Proceedings of 6th ACM Symposium on Principles of Distributed Computing, Vancouver, Canada.*
9. P.J. Courtois, F. Heymans, D.L. Parnas (1971). Concurrent control with 'readers' and 'writers'. *Communications of ACM 14,* 667-668.
10. E.W. Dijkstra (1968). Cooperating sequential processes. F. Genuys (ed.). *Programming Languages,* Academic Press.
11. Danny Dolev, Nir Shavit (1989). Bounded concurrent time-stamp systems. *Proceedings of the 21st Annual ACM Symposium on Theory of Computing, Seattle.*
12. David Gelernter (1989). The metamorphosis of information management. *Scientific American,* 54-61.
13. J. Halpern (1987). Personal Communication.
14. M.P. Herlihy (1988). Impossibility and universality results for wait-free synchronization. *Proceedings of 7th ACM Symposium on Principles of Distributed Computing, Toronto, Canada.*
15. W.D. Hillis (1985). *The Connection Machine,* MIT Press.
16. C.A.R. Hoare (1978). Communicating sequential processes. *Communications of ACM 21,* 666-677.
17. A. Israeli, Ming Li (1987). Unbounded time stamps. *Proceedings of IEEE 28th Annual Symposium on Foundations of Computer Science, New York.*
18. A. Israeli, Ming Li, Paul Vitányi (1987). *Simple Multireader Registers using Time-Stamp Schemes,* Technical Report CS-R8758, Centrum voor Wiskunde en Informatica, Department of Algorithmics and Architectures.
19. Lefteris M. Kirousis, Evangelos Kranakis, Paul Vitányi (1988). Atomic multi-reader register. Jan van Leeuwen (ed.). *Proceedings of 2nd International Workshop on Distributed Algorithms, Amsterdam, July 1987,* 278-296, Springer Verlag Lecture Notes in Computer Science, Heidelberg, Vol. 312.
20. Evangelos Kranakis (1989). Functional dependencies of variables in wait-free programs. *Proceedings of 3rd International Workshop on Distributed Algorithms, Nice, September 1989,* Springer Verlag Lecture Notes in Computer Science, Vol. 392.
21. Evangelos Kranakis, Paul Vitányi (1989). Fair, wait-free, atomic test and set. Preprint.
22. M. Loui, H. Abu-Amara (1987). Memory requirements for agreement among unreliable asynchronous processes. *Advances in Computing Research, JAI Press,* 163-183.
23. Leslie Lamport (1985). *Interprocess Communication,* Technical Report, SRI International.
24. Leslie lamport (1986). On interprocess communication, part i: basic formalism, part ii: basic algorithms. *Distributed Computing 1,* 77-101.
25. A. Lenstra (1987). Personal Communication.

26. NANCY A. LYNCH, KENNETH J. GOLDMAN (1988). *Distributed Algorithms,* Technical Report MIT/LCS/RSS 5, MIT Research Seminar Series, May 1989. Lecture Notes for 6.852.
27. NANCY A. LYNCH, MARK R. TUTTLE (1989). An introduction to input/output automata. *CWI Quarterly 2,* 217-244.
28. MING LI, JOHN TROMP, PAUL VITÁNYI (1989). *How to Share Concurrent Wait-Free Variables,* Technical Report CS-R8916, Centrum voor Wiskunde en Informatica, Department of Algorithmics and Architectures.
29. MING LI, PAUL VITÁNYI (1989). A very simple construction for atomic multiwriter register. *ICALP,* Springer Verlag Lecture Notes in Computer Science, Heidelberg.
30. L.R. MARINO (1986). General theory of metastable operation. *IEEE Transactions on Computers C-30,* 107-115.
31. LAMBERT MEERTENS (1987). Personal Communication.
32. J. MISRA (1986). Axioms for memory access in asynchronous hardware systems. *ACM Transactions on Programming Languages and Systems 8,* 142-153.
33. R. NEWMAN-WOLFE (1987). A protocol for wait-free, atomic, multi-reader shared variables. *Proceedings of 6th ACM Symposium on Principles of Distributed Computing, Vancouver, Canada.*
34. CHRISTOS PAPADIMITRIOU (1979). The serializability of concurrent database updates. *Journal of the ACM 16,* 631-653.
35. CHRISTOS PAPADIMITRIOU (1986). *Theory of Database Concurrency Control,* Computer Science Press.
36. GARY PETERSON, JAMES BURNS (1987). Concurrent reading while writing (i). *Proceedings of 6th ACM Symposium on Principles of Distributed Computing, Vancouver, Canada.*
37. GARY PETERSON, JAMES BURNS (1987). Concurrent reading while writing (ii). *Proceedings of IEEE 28th Annual Symposium on Foundations of Computer Science, New York.*
38. GARY PETERSON, JAMES BURNS (1987). *Sharp Bounds for the Concurrent Reading while Writing Problem,* Technical Report GIT-ICS-87/31, Georgia Institute of Technology.
39. GARY PETERSON (1983). Concurrent reading while writing. *ACM Transactions on Programming Languages and Systems 5,* 46-55.
40. AMIR PNUELI, LENORE ZUCK (1986). Verification of multiprocess probabilistic protocols. *Distributed Computing 1,* 53-72.
41. R. SCHAFFER (1988). *On the Correctness of Atomic Multiwriter Registers without Waiting,* Technical Report TM-364, Massachusetts Institute of Technology Laboratory for Computer Science.
42. JOHN TROMP (1989). How to construct an atomic variable. *Proceedings of 3rd International Workshop on Distributed Algorithms, Nice, September 1989,* Springer Verlag Lecture Notes in Computer Science, Vol. 392.
43. PAUL VITÁNYI, BARUCH AWERBUCH (1986). Atomic shared register access by asynchronous hardware. *Proceedings of IEEE 27th Annual Symposium on Foundations of Computer Science, Toronto.* Errata ibid 1987.

44. K. VIDYASANKAR (1985). A simple characterization of database serializability. *Proceedings of the 5th Conference on Foundations of Software Technology and Theoretical Computer Science,* Springer Verlag Lecture Notes in Computer Science, Heidelberg, Vol. 206.
45. K. VIDYASANKAR (1988). *Converting Lamport's Regular Register to Atomic Register,* Technical Report 8801, Department of Computer Science, Memorial University of Newfoundland.
46. K. VIDYASANKAR (1988). *An Elegant 1-Writer Multireader Multivalued Atomic Register,* Technical Report 8807, Department of Computer Science, Memorial University of Newfoundland.
47. K. VIDYASANKAR (1988). *Improving Peterson's Construction of 1-Writer n-Reader Multivalued Atomic Register,* Technical Report 8808, Department of Computer Science, Memorial University of Newfoundland.
48. K. VIDYASANKAR (1988). *A New 1-Writer Multireader Multivalued Atomic Register,* Technical Report 8804, Department of Computer Science, Memorial University of Newfoundland.
49. A. VAN WIJNGAARDEN, B.J. MAILLOUX, J.E.L. PECK, C.H.A. KOSTER (1969). Report on the algorithmic language algol68. *Numerische Mathematik 14,* 79-218.