Centrum voor Wiskunde en Informatica

Centre for Mathematics and Computer Science

J.T. Tromp

How to construct an atomic variable

Computer Science/Department of Algorithmics & Architecture Report CS-R8939 October

1989

Centrum voor Wiskunde en Informatica
Centre for Mathematics and Computer Science

J.T. Tromp

How to construct an atomic variable

Computer Science/Department of Algorithmics & Architecture Report CS-R8939 October

---

The Centre for Mathematics and Computer Science is a research institute of the Stichting Mathematisch Centrum, which was founded on February 11, 1946, as a nonprofit institution aiming at the promotion of mathematics, computer science, and their applications. It is sponsored by the Dutch Government through the Netherlands Organization for the Advancement of Research (N.W.O.).

How to Construct an Atomic Variable

John Tromp
Centrum voor Wiskunde en Informatica
Kruislaan 413, 1098 SJ Amsterdam, The Netherlands
Email: tromp@cwi.nl

October 11, 1989

Abstract

We present solutions to the problem of simulating an atomic single-reader, single-writer variable with non-atomic bits. The first construction, for the case of a 2-valued atomic variable (bit), achieves the minimal number of non-atomic bits needed. The main construction of a multi-bit variable avoids repeated writing (resp. reading) of the value in a single write (resp. read) action on the simulated atomic variable. It improves on existing solutions of that type in simplicity and in the number of non-atomic bits used, both in presence and in accesses per read/write action. We show how to verify these constructions by machine, based on atomicity-testing automata.

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,
safe bits, handshake, tracks, verification, automata.
Note: This paper is submitted for publication elsewhere.

1 Introduction.

Communication plays a vital role in any distributed system, allowing multiple processors to share and exchange information. Conventionally this communication is based on mutually exclusive access to a shared *variable*. This is the case not only in a shared memory system, but also at the two endpoints of a link in a message based system. Unfortunately, this exclusive nature of access may force a user of such a variable to wait for another one and therefore impedes the parrallellism inherent in distributed systems. In the last years interest has focussed on *wait-free* variables, which can be accessed concurrently without any form of waiting. The question is how to construct such variables in terms of lower-level hardware, like flip-flops.

Peterson was one of the first to investigate this question in [8], giving a construction for a single-writer, multi-reader, multi-bit atomic variable from single-writer, multi-reader atomic bits. Later, Lamport [3], sparked off interest in the subject by developing a precise theory and formalisms—apart from presenting some solutions to subproblems. It is worth noting that most papers use the word "register" instead of variable.

The ultimate goal is to build a variable accessible to any fixed number of users—each having write and read capabilities—which can hold any fixed number of values, and whose accesses behave atomically. The latter means that for any sequence of operations on the variable, the partial precedence order (defined later) among those operations must have a total extension (for external consistency) such that each read operation returns the value from the write operation which is the last to precede it in the total order (for internal consistentency).

Paper	used	write	read
[8]	$3b + 10$	$3b + 13$	$3b + 23$
[4]	$4b + 39$	$b + 5$	$b + 26$
this	$4b + 8$	$b + 2$	$b + 4$

Table 1: worst case number of bits

The construction of this "ultimate" variable is not done directly from the most primitive kind of variable. Rather, this task is more conveniently split into two subtasks: the construction of atomic, multi-bit, single-writer, single-reader variables and next the construction of atomic, multi-bit, multi-user variables from the former type.

This partition can be justified by the nature of the problems involved. In the first construction (as in [3]), the multi-bit value is to be *distributed* over a multiple number of bits. In the second construction (as in [11]), the value of the multi-user variable is *replicated* among all users[1] along with control information which allows each user to identify the most recently written value. This is the more complex problem, as witnessed by the fact that many proposed (and often proven) solutions were later found to be erroneous. The interested reader is referred to [1, 5, 7, 10, 11].

In this paper we attack the first problem, and also give special attention to the case of constructing a single-bit atomic variable.

2 Comparison with Related Work

There are basically two approaches that can be taken in order to construct a multi-bit variable from a linear number of single bits. The first was taken by G.L. Peterson in [8] and involves keeping 3 copies of the multi-bit value, called the *tracks* (in the original paper, they are called *buffers*). Apart from the 3 tracks, there are some *control* bits which we collectively call the *switch*. In this approach the writer writes the new value to all three tracks. The reader reads from all tracks, but in a different order. The switch allows the reader to determine which track was read without interference from the writer.

In Kirousis et. al. [4], the second approach was taken. The idea is that the writer and the reader access only a single track, and that the switch ensures that they never access the same track simultaneously. The price to be paid for the reduced number of track-accesses is the necessity of using four tracks.

In this paper a simplification of the latter construction is presented. Table 2 gives a comparison of these constructions for a b-bit atomic variable. The "used" column displays the total number of safe bits used in the construction ("space complexity"). The "write" ("read") column gives the worst case number of safe bits that must be accessed in a write (read) action on the atomic variable. A trade-off between time and space is clearly visible.

We also present a solution to the special case of constructing an atomic bit with a minimal number of non-atomic bits. This problem was solved independently and earlier by J. Burns and G. Peterson [9], using a very similar construction. The use of finite state machines for (automated) correctness verification is related to the work by Clarke and Emerson (e.g. [2]) and is new to this area.

3 Preliminaries

In this paper we consider variables which can be written by one user, called the *writer*, and read by another, the *reader*. Both users may be accessing the variable concurrently without ever having

[1] to have a communication path between any pair of users, we need to maintain $\Omega(n^2)$ copies.

to wait for one another. This means that no assumptions are made about the relative speed of the users, and that the correct operation of the variable is not impaired by halting either one. As stated in the introduction, we aim to construct an atomic, multi-bit, single-writer, single-reader variable. The objects we use in this construction are *safe*, single-bit, single-writer, single-reader variables, or simply *safe bits*. These are the mathematical counterparts of flip-flops, in the sense that real-life flip-flops can be argued to satisfy the *safety* property. Before giving rigorous definitions for the notions of safe and atomic, we first state some preliminary definitions.

In order to distinguish the accesses to the constructed atomic variable (the *higher level*) from the accesses to the safe bits (the *lower level*), we call the former *actions*, and the latter *subactions*. As we will see, each higher-level action is composed of a number of subactions—where the *wait-free* condition requires this number to be bounded.

Let V be a variable and $\mathcal{A}$ (the set of accesses) the union of a set of writes $\mathcal{W}$ to V and a set of reads $\mathcal{R}$ from V. The result of a read is a value which is said to be *returned* by that read. Each access $a \in \mathcal{A}$ occupies a *time interval* $(s(a), f(a))$, where $s(a)$ is the start time and $f(a)$ the finish time of access a. All start and finish times are assumed to be pairwise distinct. We define a precedence relation $\rightarrow$ on $\mathcal{A}$ as follows: $a \rightarrow b$ iff $f(a) < s(b)$. We say that a *overlaps* b, or a and b overlap, if they are $\rightarrow$-incomparable, that is, $a \not\rightarrow b \wedge b \not\rightarrow a$. *Complete overlap* of a by b means that $s(b) < s(a) < f(a) < f(b)$. We assume that the set $\{w | w \rightarrow r\}$ is finite for any read r. We call the pair $(\mathcal{A}, \rightarrow)$ a *run*. The writes in $\mathcal{W}$ are totally ordered by $\rightarrow$, and so are the reads in $\mathcal{R}$, in accordance with the requirement that a user can only perform one access at a time.

We relate the reads to the writes in terms of a reading function. A partial function $\pi : \mathcal{R} \rightarrow \mathcal{W}$ is a reading function if for every read $r \in \mathcal{R}$ on which π is defined, $\pi(r)$ writes to V the value returned by r. Unless explicitly stated, the reading function will be total (non-total reading functions will be needed in the definition of safety). We call the triple $(\mathcal{A}, \rightarrow, \pi)$ a *system execution.*

We can now define atomicity:

Definition 1 *A system execution $\sigma = (\mathcal{A}, \rightarrow, \pi)$ of the variable V is atomic iff there is a total extension $\rightarrow'$ of $\rightarrow$ consistent with π, i.e. for every read $r \in \mathcal{R}$, $\pi(r)$ is the last write preceding r in the total order $\rightarrow'$.*

In the case of a single writer, a simplification of the general definition above can be given which avoids the use of a total ordering, [3, 7, 6]:

Definition 2 *A system execution $\sigma = (\mathcal{A}, \rightarrow, \pi)$ of the single-writer variable V is atomic iff the following three properties hold for all $r, r_1, r_2 \in \mathcal{R}$ and $w \in \mathcal{W}$:*

A0 *not* $(r \rightarrow \pi(r))$

A1 *if* $r_1 \rightarrow r_2$, *then not* $(\pi(r_2) \rightarrow \pi(r_1))$

A2 *not* $(\pi(r) \rightarrow w \rightarrow r)$

Equivalence of definitions 1 and 2 is shown by proving ([3])

$$\exists \text{consistent total extension } \rightarrow' \Leftrightarrow \rightarrow \text{ satisfies } A0, A1, A2$$

First of all, if any of the three conditions on $\rightarrow$ is violated, then it will clearly be impossible to find a consistent extension $\rightarrow'$, thus proving the first direction. To prove the converse, we construct the following $\rightarrow'$: Merge the reads into the totally ordered set of writes, such that each write w is immediately followed by the reads r with $\pi(r) = w$. Naturally, this order is consistent with π. By A1, we can merge while preserving the original total ordering among the reads. A0 and A2 ensure that the precedence between a read and a write is also extended. If $w \rightarrow r$, then by A2, $\neg(\pi(r) \rightarrow w)$, so by the construction of $\rightarrow'$ we have $w \rightarrow' r$. If $r \rightarrow w$, then by A0, $\pi(r) \rightarrow w$, so again by the construction of $\rightarrow'$ we have $r \rightarrow' w$. □

Thus, in atomic runs, the partially ordered set of accesses can be linearized while respecting the logical read/write order. In addition to definition 2, we have:

Definition 3 *σ is* regular *iff A0 and A2 hold. σ is safe iff $(\mathcal{A} - \mathcal{R}', \rightarrow, \pi)$ is atomic, where $\mathcal{R}' = \{r \in \mathcal{R} | \exists w \in \mathcal{W}(w \not\rightarrow r \wedge r \not\rightarrow w)\}$ is the set of reads which overlap a write (in which case π is left undefined).*

Thus, in a safe run, a read overlapping a write may return any value in the domain of the variable. The other actions will then be totally ordered, such that each non-overlapping read returns the value written by the last preceding write.

Definition 4 *Variable V is atomic (regular,safe) iff for each of its runs $(\mathcal{A}, \rightarrow)$, there exists a reading function π, such that the system execution $(\mathcal{A}, \rightarrow, \pi)$ is atomic (regular,safe).*

It is clear from these definitions that an atomic variable is regular, and that a regular variable is safe. We call a reading function *normal* if it satisfies A0, i.e. it doesn't map a read to a write which starts after the read finishes. In practice, only normal reading functions are considered.

3.1 Making Safe Bits Regular

Consider a safe bit V which is not regular. For such bits there exists a (safe) run $(\mathcal{A}, \rightarrow)$ such that for all π, $(\mathcal{A}, \rightarrow, \pi)$ is irregular. This must be caused by a read r, which cannot be mapped to a write $\pi(r)$ without violating either A0 or A2. By the safety of V, we have $r \in \mathcal{R}' \neq \mathcal{R}$, hence r overlaps a write $w \in \mathcal{W}$. If w changes the value of V (from 0 to 1 or from 1 to 0), then r can be mapped to either w or the previous write, depending on which one wrote the same value as r returned. This would clearly satisfy both A0 and A2. So w doesn't change the value of V, e.g., the following run is safe and irregular: a write of 0, followed by another write of 0 which completely overlaps a read of the value 1.

Hence we have shown that a safe bit can be irregular if a read overlaps a non-changing write.

To make a safe bit regular, it therefore suffices never to write to the safe bit its current value, i.e. to only *change* the bit. This is the approach taken in this paper—instead of writing a boolean value to a safe bit, we only allow the writer to change it, thereby making the bit a regular one. A similar solution was first given by Lamport [3].

3.2 Safe Bits can Flicker

Obviously a safe bit is a weaker type of bit than an atomic one in that its system executions may violate property A1. Assume the occurrence of such a violation, say, with $r_1 \rightarrow r_2$ and $\pi(r_2) \rightarrow \pi(r_1)$. By A0 we have that $s(\pi(r_1)) < f(r_1)$. Also, $s(r_2) < f(\pi(r_1))$, since otherwise $\pi(r_2) \rightarrow \pi(r_1) \rightarrow r_2$, violating A2. Finally, since $s(\pi(r_1)) < f(r_1) < s(r_2)$, $\pi(r_2)$ is the immediate predecessor of $\pi(r_1)$. We see that the two reads both overlap the write $\pi(r_1)$. The former read returns the new value, while the latter returns the old value. That is why we refer to a violation of A1 as a *new-old inversion*.

The behaviour of a safe bit can be described by saying that its value may *flicker* during a write and only when the write finishes does it stabilize to the complementary value of that at the start of the write. This contrasts with the behaviour of an atomic bit, which appears to change its value in a single, indivisible time instant, somewhere between the start and finish of the write.

4 Problem Statement

It remains to define what it means to construct an atomic variable from safe bits. (Recall that in this paper all variables are single-writer, single-reader, hence two users). Such a construction is defined by an *architecture* and a pair of *protocols*, one for each user. The architecture specifies the number of safe bits, their names and how they are connected among the two users. Each safe bit can be connected to the reader and the writer in one of only two ways: changed by the writer and read by the reader, or changed by the reader and read by the writer. The user that can change a bit is said to be the *owner*.

A protocol specifies how the writer (reader) can change (read) the atomic variable in terms of changes to and reads from the safe bits. In addition the protocols may make use of *local variables*, which can be viewed as safe bits that are changed and read by the same user. These are however not considered part of the architecture, which specifies only *shared* bits.

We consider only *wait-free* protocols, i.e., the number of safe bit accesses in a single protocol execution must be bounded by a fixed constant. This requirement forbids solutions in which a user might have to wait for a safe bit to change value.

A read or write action on the atomic variable consists of an execution of the corresponding protocol. We use the terms "action" and "protocol execution" interchangeably. A construction is initialized by an initial write that sets the atomic variable to the value 0. This allows the definition of a reading function on every read action. All other shared bits and local variables are also initialized to 0. Finally, each run of the construction must satisfy the atomicity criterion.

In the next section we consider the special case of a 2-valued atomic variable. After proving that 3 safe bits are needed to construct an atomic bit, we develop a construction that achieves this lower bound, followed by a proof of correctness. The general case of a b-bit (2^b-valued) atomic variable is dealt with in section 6.

5 Optimal Construction of Atomic Bits

5.1 A Lower Bound on the Number of Safe Bits needed to Construct an Atomic Bit

We demonstrate that 3 safe bits are required in a construction of an atomic bit, in particular, 2 bits owned by the writer, and 1 owned by the reader.

The reason that a single bit (call it V), owned by the writer doesn't suffice, is exemplified by a run $(\mathcal{A}, \rightarrow, \pi)$, where $\mathcal{A} = \{w, r_1, r_2\}$, and $\rightarrow = \{(r_1, r_2)\}$, i.e. write w (which changes the atomic bit from its initial value 0 to 1), overlaps two reads r_1 and r_2. Without loss of generality, we can assume that the writer changes V at least once during the execution of its protocol. We may also assume that, depending on the values obtained by reading V, but independent of the values of local variables, the reader protocol can return both a 0 and a 1. Consider now the case when both reads occur during the first time that w changes V. Then by the flickering of V, it is possible for r_1 to return 1 while r_2 returns 0. But now there is no reading function which satisfies all three atomicity conditions.

The example shows that a communication channel from the writer to the reader of only one safe bit is too narrow—at least 2 safe bits are necessary.

In [3], Lamport has shown the necessity of two-way communication, i.e., the reader must own at least 1 bit. We rephrase his proof.

Assume there are n safe bits in the architecture, all owned by the writer. They can be in one of 2^n states, denoted $0, 1, \ldots, 2^n - 1$. In every write, the writer changes some safe bits, which corresponds to a *state sequence*, the sequence of states through which the safe bits move. I.e., each change of a safe bit involves a transition from one state to another. By the wait-free condition, the number of safe bit changes in each single execution of the writer protocol must be bounded by a fixed constant, say k, hence the length of the state sequences is bounded by $k+1$ and the number of state sequences by $(2^n)^1 + \cdots + (2^n)^{k+1} = \frac{(2^n)^{k+2} - 2^n}{2^n - 1}$. Therefore, there must be a state sequence $S = s_0, \ldots, s_l$ with $l \leq k$, such that (with an infinite set of writes) the writer infinitely often changes the atomic bit from 0 to 1 according to this state sequence.

Consider a set $\mathcal{A} = \mathcal{W} \cup \mathcal{R}$ of actions, where $\mathcal{R}$ consists of $l+1$ consecutive read actions $r_0 \rightarrow r_1 \rightarrow \cdots \rightarrow r_l$, and $\mathcal{W}$ contains $l+1$ (non-consecutive) writes $w_0 \rightarrow w_1 \rightarrow \cdots \rightarrow w_l$ which change the atomic bit from 0 to 1 according to state sequence S. Assume that r_i "sees" the safe bits in state s_{l-i}, thus fixing the values returned by the reads. The runs we consider below will all be consistent with this assumption. We will prove by induction that all r_i must return 1.

Base: Since r_0 sees the safe bits in state s_l and could have ocurred just after w_0, it must return the value written by w_0, which is 1.

Induction step: Consider the following run: for all $j < i$ and $j > i+1$, r_j occurs while w_j has reached state s_{l-j} (that is, after w_j has changed $l-j$ safe bits). Furthermore, both r_i and r_{i+1} occur while w_i is changing the state from $s_{l-(i+1)}$ to s_{l-i}. By induction r_i returns 1, the new value of write w_i. For this run to be atomic, hence to avoid new-old inversions, r_{i+1} must also return 1.

So r_l must return 1. But this invalidates the atomicity of the following run: for all $j < l$, r_j occurs while w_j has state s_{l-j}, and r_l occurs just before w_l, with the safe bits in state s_0. In this run r_l immediately follows a write of the value 0, hence 1 is an invalid return value. □

We conclude that an atomic bit cannot be simulated with only one-way communication. The sequence of safe bits changed during the protocol execution of the writer, must therefore depend on information that it receives from the reader. We have thus proved a lower bound on the number of safe bits needed in a construction:

Lemma 1 *A construction of an atomic bit from non-atomic, safe bits requires at least* 2 *bits owned by the writer, and at least* 1 *bit owned by the reader.*

The next few sections deal with the development of a solution, which will be derived by gradual refinement starting from a trivial but incorrect solution. After each proposed solution, we present a counter-example which highlights the remaining shortcomings. Let us first describe the architecture of the construction.

5.2 The Architecture

We aim to attain the optimal number of shared safe bits, which is 3. As shown above, the reader will be the owner of one of these, so we simply call that bit "R." One of the two bits owned by the writer will be used to hold the value of the simulated atomic bit, we therefore call it "V." For the other bit owned by the writer, we consider "W" to be an appropriate name. To sum up, we have the following 3 safe bits:

5.3 A Trivially Incorrect Solution

In the protocols, we make use of the following statements. The owner of a safe bit B can execute the statement "`change B`" to change its value. Remember that during this change, the value may flicker between 0 and 1. Local variables have lower case names to distinguish them from the shared bits. In all the protocols presented here, the local variables are 2-valued (bits), and are used to hold a copy of V. For this purpose there is a statement "`read loc := V`," whose effect is to read V and store the result in the local variable *loc*. Given the safety of a shared bit, we know that the changes to and reads from it obey the conditions A0 and A2 (see definition 4), for some reading function. The next statement we consider is the conditional "`if` test `then` statement." There is no `else` part, because it just so happens that we don't need it in any of our protocols. The semantics are obvious—if the test succeeds, then the statement is executed, otherwise it is skipped. The test is either "`W==R`"[2] or "`W<>R`." Performing such a test is done by first reading the flag owned by the other user (e.g. the writer reads R in its test). This read is implicit in the test, and is not stated explicitly in the protocol as a separate `read` statement. In order to be able to compare this value against the value of one's own flag, we assume that the owner of a flag keeps track of its value. This

[2] This notation for equality comes from the language C and helps to line up the statements.

abbreviated notation for tests will prove to make the protocols more concise and readable. The final statement in our repertoire is "`return loc`," with *loc* again a local bit. It is used by the reader to exit the execution of its protocol and to specify the return value.

We can now state the first pair of protocols:

```
WRITER PROTOCOL                 READER PROTOCOL

change V                        read v := V
                                return v
```

Note that bits W and R are not yet used. We rephrase the example of section 5.1. Write actions are denoted by w_i and read actions by r_i, where i numbers the actions of each user.

w_1 starts changing V from 0 to 1
r_1 reads $v := 1$ and returns the new value
r_2 reads $v := 0$ and returns the old value

The initializing write action is called w_0 for convenience. Since only two write actions have occurred, the reading function π must map each read action to the write action that wrote the returned value. Thus $\pi(r_1) = w_1$ and $\pi(r_2) = w_0$. But since $r_1 \rightarrow r_2$ and $w_0 \rightarrow w_1$, this run violates A1 and is therefore not atomic.

5.4 Handshaking

If multiple read actions occur while V is being changed by the writer, then the value returned by the later actions should be based not only on the values obtained from reading V.

One solution is to detect occurrence of this event, in which case the reader could return the same value as it did before (to prevent new-old inversions). This detection is possible by exploiting the fact that while the writer is changing V, it is not changing W. So if two consecutive read actions notice the same value of W, then the second would like to return the same value as did the first. We call this value the previous return value, or simply the *previous value*.

For this purpose we give the reader a local variable called v, which is to hold the value that the reader returns (or a more recent value, as we will later see). The writer would like to change W as often as possible so that the reader will notice the change and obtain the value most recently written to V. On the other hand, if W is changed too often, then the reader will not notice every single change and may get the impression that W didn't change at all, while it actually changed more than once. This explains why we need the third safe bit, R. It is needed by the reader to tell the writer when it has seen a change in W.

In summary, what we need is a *handshake* system, in which each user decides to change its flag once it sees a change in the other user's flag. This is equivalent to saying that one user tries to make the flags equal while the other one tries to make them different.

We (arbitrarily) choose to have the writer make W different from R, and have the reader make R equal to W. So the writer protocol will have a *handshake* statement "`if W==R then change W`," and the reader protocol a handshake statement "`if W<>R then change R`." We argued above that the reader protocol should also have statements "`if W==R then return v`" and "`read v := V`" occurring in that order. And of course the writer protocol is not complete without the statement "change `V`."

5.5 Ordering the Writer Protocol

We first try to determine the relative order of the two statements that are required in the writer protocol. We already explained that the purpose of the writer handshake is to inform the reader that the value of the simulated atomic bit has changed. Then intuitively, the handshake statement

should come *after* the statement changing V. We will give an example to rule out the reverse order, in which the writer protocol consists of a `if W==R then change W`, followed by a `change V`. The only assumption we make about the reader protocol is that it contains the three statements given earlier. Consider the following run:

w_1 executes `if W==R then change W` and changes W to 1
r_1 changes R to 1 and reads $v := 0$ (in whatever order)
w_1 changes V to 1 and finishes
r_2 passes the test `W==R` and returns $v = 0$

Clearly, the second read doesn't return the most recent value—condition A2 is violated for normal reading functions. It appears that the *only* way to surmount this problem, is to have a handshake occur *after* the change of V.

Hence we adopt the following, intuitively sound, writer protocol:

```
WRITER PROTOCOL

change V
if W==R then change W
```

As it happens, no more changes will be required in the writer protocol—this is the final one.

5.6 Ordering the Reader Protocol

As in the writer's case we also have to determine where to position the `read v := V` in the reader's protocol relative to the handshake `if W<>R then change R`. We will analyze both options in turn.

Case 1. No handshake occurs after the read of V.

The problem with this choice is that the W, R flags may turn out different after the read of V. The following run shows that this can happen even while the writer is busy changing V, thus causing a new-old inversion. Assume that the reader protocol has the `if W<>R then change R` statement before the `read v := V` statement.

w_1 changes V from 0 to 1, sees $R = 0$ and starts changing W from 0 to 1
r_1 sees $W = 1$, changes R to 1, reads V and returns $v = 1$
r_2 sees $W = 0$ and changes R to 0
w_1 finishes with $W = 1$
w_2 starts changing V from 1 to 0
r_2 reads V and returns $v = 0$
r_3 sees $W = 1$, changes R to 1, reads V and returns $v = 1$

Any reading function must map r_1 and r_3 to w_1 since only w_1 changed the atomic bit to their return value 1. But the read action in between (r_2) returned 0! Clearly, any mapping of r_2 violates A1.

Case 2. A handshake occurs after the read of V.

Unfortunately this option also has a problem. It is now conceivable that the writer executes a complete write action after the read of V but before the handshake. Consider the following run:

r_1 reads V
w_1 changes V from 0 to 1, sees $R = 0$, changes W to 1 and finishes
r_1 sees $W = 1$, changes R to 1 and returns $v = 0$
r_2 sees $W = 1$ and returns $v = 0$

The reading function must map r_2 to w_0 and hence violates A2. It is clear that the writer's handshake is "masked" by that of the reader, resulting in a later read action returning an out-of-date value. From this counter example we draw the following conclusion:

Rule 1 *The reader shouldn't change R between reading a value and using it in the next read action as the returnvalue.*

5.7 Performing Extra Reads

The above two cases create a dilemma in the following sense: After the reader reads V, it might check to see whether W equals R. If they are seen to be equal, then neither of the two problems described above can occur. If however the flags are seen to be different then by rule 1 the reader cannot change R and by the first case in section 5.6 it cannot return the value it just read.

In order to rectify this situation, we have to retract our assumption that the reader protocol performs only one read of V and does only one handshake. The most obvious approach is to repeatedly read V and do a handshake until the two flags are seen to be equal. However, the wait-free condition requires that the number of repetitions must be bounded by a constant. What do we gain by repetition ? Well, suppose the reader sees $W \neq R$, changes R and later sees $W \neq R$ again. Then it has gained the knowledge that the writer was changing W, and hence that V wasn't flickering all that time. This knowledge can be used to avoid new-old inversions. In particular, if the reader were to read V and later on in the same protocol execution it would notice a change in W, then it can safely return the value that was read, without fear of returning an older value in a later read action. After all, a new old inversion can only occur if the two reads of V both overlap a change in V by the writer, in which case W doesn't change between the reads.

This observation leads us to the following reader protocol. Before doing a handshake followed by a `read v := V`, it makes an *extra* read of V into the new local variable x. Afterwards, it again tests equality of W and R. If this test succeeds, then it simply forgets about the extra read, and returns v. If, however, the flags are different, then it decides to use the extra read and returns x instead.

```
READER PROTOCOL

if W==R then return v
read x := V
if W<>R then change R
read v := V
if W==R then return v
return x
```

In contrast with v, the value of x need not be remembered between succesive executions of the reader protocol.

The read into x must come after the first line. Otherwise, if one read action were to return v, and the next one to return x, these values may constitute a new-old inversion.

There is still a (minor) problem with this protocol. We argued above that returning x is safe because W was changed afterwards and so any value read after that is free from new-old inversions. But the next read action may return in the first line of the protocol with the value of v, and because this v was read before noticing the change in W, it may still be older than x. This problem clearly stems from the fact that the previous value is not well defined when x is returned. A straightforward alternative is to assign x to v before returning x. Unfortunately, as pointed out by rule 1, x is not a valid previous value in case R was changed in the third line. We need a value that is known to be recent. We therefore solve this problem by inserting another `read v := V` statement just before the return of x.

We have now arrived at the final protocols:

```
WRITER PROTOCOL                 READER PROTOCOL

change V                        1.  if W==R then return v
if W==R then change W           2.  read x := V
```

```
3.  if W<>R then change R
4.  read v := V
5.  if W==R then return v
6.  read v := V
7.  return x
```

5.8 Proof of Correctness

Let $(\mathcal{A}, \rightarrow)$ be a run of the atomic bit. Then we can find a lower level run for each of the three safe bits, consisting of all the accesses to that safe bit and the precedence relation defined from their start and finish times. Let π' be the reading function that makes the run on V safe. Let $\mathcal{W}$ be the set of write actions in $\mathcal{A}$, and $\mathcal{R}$ the set of read actions in $\mathcal{A}$. We must prove the atomicity of $\sigma = (\mathcal{A}, \rightarrow, \pi)$ for some reading function π. We define π in the natural way as follows. Let $r \in \mathcal{R}$ be any read action and let *loc* be the local variable returned by r. We can define ρ, the subread of r's return value, as the last subread from V into *loc* before r returns. E.g. if r returns in line 7 then ρ is the read in line 2 of r, and if r returns in line 1, then ρ is the read in line 4 of some earlier read action. Let w be the write action which executed $\pi'(\rho)$ in the first line of its protocol. Then we define $\pi(r) = w$.

Proof of A0 Intuitively, since the underlying bits are safe, a read action can only return the value of a past or concurrent (overlapping) write action. We formally prove A0 by contradiction: Assume that for some $r \in \mathcal{R}$, $r \rightarrow \pi(r)$. Let ρ be the subread of r's return value as above. Then $f(\rho) < f(r)$ and by definition of π, $s(\pi(r)) < s(\pi'(\rho))$. Together these imply that $\rho \rightarrow \pi'(\rho)$, which contradicts the safety of π'.

Proof of A1 The proof is again by contradiction. Let $r_1, r_2 \in \mathcal{R}$ be such that $r_1 \rightarrow r_2$ and $\pi(r_2) \rightarrow \pi(r_1)$. We assume without loss of generality that r_1 is the first such read action. Let $\rho_i, i \in \{1, 2\}$ be the subread from V of r_i's return value. For notational convenience, we use the element-of-set symbol $\in$ to denote that a safe bit access is part of a read or write action. Then $\rho_1 \in r_1$ if r_1 returned in line 5 or line 7. Otherwise, if r_1 returned in line 1, then by the minimality of r_1, ρ_1 is in the immediately preceding read action. Defining ω_i as $\pi'(\rho_i)$, we also have $\omega_i \in \pi(r_i)$. This clearly implies that $\omega_2 \rightarrow \omega_1$. According to the reader protocol and because $\pi(r_1) \neq \pi(r_2)$ implies $\rho_1 \neq \rho_2$, we have that $\rho_1 \rightarrow \rho_2$. For this new-old inversion to occur on safe bit V, it is necessary that $s(\omega_1) < f(\rho_1)$ (to satisfy A0) and that $s(\rho_2) < f(\omega_1)$ (to satisfy A2). This means that

$$\text{the value of } W \text{ doesn't change between } f(\rho_1) \text{ and } s(\rho_2). \tag{1}$$

We now consider all three possible cases of the position of ρ_1.

`read x := V` in line 2 Then r_1 returned in line 7 of its protocol execution, so in line 5, it sees W different from R. But in line 3 R is made equal to W. Because ρ_2 is either the read in line 6 of this protocol execution, or a later read, we have found a contradiction with (1) above.

`read v := V` in line 4 Then r_1 returned in line 5 of its protocol execution, after seeing W equal to R. Since $\rho_1 \rightarrow \rho_2$, ρ_2 must be part of some later read action which sees W different from R in line 1 of its protocol execution. This contradicts (1) again.

`read v := V` in line 6 Then r_1 returned in line 1 of its protocol execution (which immediately succeeds that of ρ_1) after seeing W equal to R. This case therefore reduces to the previous one.

We have shown that the assumed violation of A1 leads to a contradiction.

Proof of A2 The proof is once again by contradiction. Let $r \in \mathcal{R}, w \in \mathcal{W}$ be such that $\pi(r) \rightarrow w \rightarrow r$. Let ρ be the read from V of r's return value as usual, and ω the write to V in w. From $\pi'(\rho) \in \pi(r)$ and $\omega \in w$ follows $\pi'(\rho) \rightarrow \omega$. By the safety of V, $\neg(\omega \rightarrow \rho)$, in other words, $s(\rho) < f(\omega)$. Hence $\rho \notin r$, and r must have returned in line 1 of its protocol execution after seeing W equal to R. After ω however, write action w makes sure that W is different from R. So R must

have been changed between ρ and r. According to the reader protocol, this is done in line 4, and is followed by a `read v := V` statement. This read between ρ and r contradicts the definition of ρ. This completes the proof. □

Lemma 1 and the given construction prove the following

Theorem 1 3 *safe bits are necessary and sufficient to construct a single-reader, single-writer, atomic bit.*

6 The 4-track Protocol

We return to the general problem of constructing a b-bit atomic variable with a linear number of safe bits. These can be divided into some *control bits*, collectively called the *switch*, and several b-bit tracks, whose purpose is to hold the values of the atomic variable, We start out with an explanation of why 4 tracks should be necessary (no *proof* is known to me). Remember that we want the tracks to be *collision-free*, i.e., the writer and the reader must never access the same track simultaneously. Since the accesses to a collision-free track are, by definition, serialized, it follows that

Lemma 2 *A collision-free track of b safe bits is an atomic b-bit variable.*

The problem is that in order to make a track collision-free, we must employ a multitude of them. In particular, the following scenario exemplifies that at least 4 tracks are required to avoid collisions.

In the first write action, the writer writes the first track and changes the switch accordingly. In the second write action, the writer writes the second track and while it changes the switch, the reader decides, by accessing the switch, which track to read. The idea is that the reader chooses either the first or the second track, but that the writer will have no way of knowing the outcome of the choice. So even if the reader intends to make its choice visible in the switch, then it hasn't yet done so at this point. Now the writer executes a third write action and will obviously have to go to a third track, which is again followed by the appropriate changes to the switch. Finally the writer starts a fourth write action. It might be able to see that the reader has started a read and that its choice is limited to the first two tracks. To avoid collisions it cannot write on either of these two tracks. But track three is forbidden too, since at any time the reader could finish its read and start a new one, which would have to be from track three.

It remains to show that 4 tracks suffice.

We conveniently split the 4 tracks into 2 groups T_0, T_1 of 2 tracks $T_{i,0}, T_{i,1}$ each. In order to avoid collisions, the writer always tries to go to the group other than where it sees the reader. The reader in turn wants recent values, hence it tries to go to the group where it sees the writer. Both the reader and the writer use part of the switch to signal the other user about the group they are in. For the moment this involves an atomic bit W for the writer, and an atomic bit R for the reader. In addition, the switch has two *trackdisplays* D_0, D_1, one for each group, displaying the most recently completed track. For the moment, these too are atomic bits. Later we will show how to use safe bits instead. Now when the writer completes a write action, the new value will be on track T_{W,D_W}.

In summary, the architecture consists of 4 tracks of b safe bits each and the following 4 atomic bits, which comprise the switch:

We can now informally state the writer protocol. The writer starts by reading R, the group that the reader is in, and compares it to W, the writer's group. If they are equal, then the reader must have left the other group, so the writer simply writes to a track in that other group, and changes W afterwards. It chooses the displayed track so that it doesn't have to change the trackdisplay. If R is different from W, then the writer writes to the other track in its group and changes the trackdisplay D_W afterwards.

The reader protocol is then as follows: The reader starts by reading W to see if the writer has vacated the reader's group. In that case the reader changes R and follows the writer to the other group.

Next, the reader reads the trackdisplay D_R of its group. It then reads the track T_{R,D_R} and returns the obtained value.

In a programming language, not unlike the one introduced in section 5.3, the above protocols look like:

```
WRITER PROTOCOL                  READER PROTOCOL

1.  if R==w then                 if W<>r then
2.    w := 1-w                     r := 1-r
3.    write track T[w,d[w]]        change R
4.    change W                   endif
5.  else                         read d := D[r]
6.    d[w] := 1-d[w]             read track T[r,d]
7.    write track T[w,d[w]]
8.    change D[w]
9.  endif
```

The lower-case local variables hold copies of the similarly named shared bits. An array notation is used for the tracks and displays instead of the index notation that we reserve for the text. The access to a track has been compressed to a single statement since we can ignore how many bits must be changed and in what order. For notational convenience, we do not mention the value to be written in the writer protocol or the value to be read in the reader protocol. Since each protocol execution involves exactly one track access, the meaning should be obvious.

Consider a run of the above construction. Each action contains lower-level accesses to the atomic bits of the switch and to the safe bits of a track. By definition 1, the partial order on the accesses to each atomic bit can be extended to a total one. Intuitively, the accesses to different atomic bits can then also be totally ordered. In [6], it was shown that this is indeed the case, if the precedence relation is defined in terms of a global time[3]. Using this total ordering on all atomic bit accesses, we can model a run by a sequence of *state transitions*, each transition corresponding to an atomic bit access. In this model, the states of the writer are:

0 idle, i.e., before the atomic read of R in line 1,

1 between the atomic read of R and the atomic change of W in line 4, when it is writing track $T_{1-W,D_{1-W}}$,

2 between the atomic read of R and the atomic change of D_W in line 8, when it is writing track $T_{W,1-D_W}$.

Thus, the writer is always moving from state 0 to either state 1 or state 2 (depending on the outcome of the test), and then back to state 0. The states of the reader are:

0 idle, i.e., before the atomic read of W in line 1,

1 between the atomic read of W and the atomic change of R in line 3,

[3] This *global time assumption* is equivalent to the *interval axiom*: if $a \rightarrow b \wedge c \rightarrow d$, then $a \rightarrow d$ or $c \rightarrow b$.

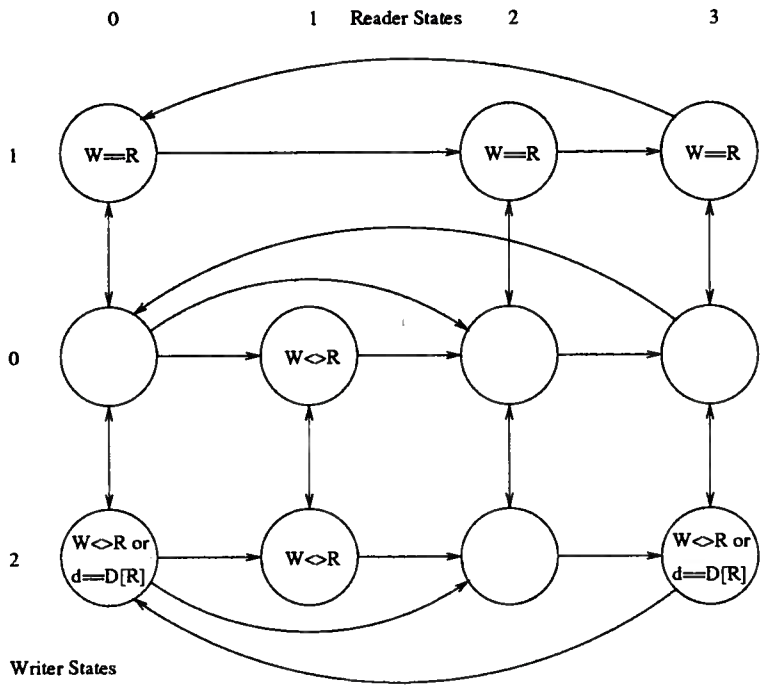

Figure 1: state diagram of 4-track construction

2 just before the atomic read of D_R in line 5,

3 after the atomic read of D_R, when it is reading track $T_{R,d}$.

Thus the reader is always moving from state 0 to either state 1 and then to state 2 or directly to state 2, then on to state 3, and finally back to state 0.

Now figure 1 shows all possible transitions in a run of the 4-track construction. It can be easily checked that the invariants in the nodes hold. Note that it is impossible for the writer and the reader to be in state 1 simultaneously.

Lemma 3 *The 4-track construction is collision-free.*

Proof. We denote the combined writer and reader state in a pair (ws, rs). Collisions can only occur in states $(1,3)$ and $(2,3)$, when both the writer and the reader are accessing a track.

In the former case, the writer is in group $w = 1 - W$, while the reader is in group $r = R$. From the diagram we see that $W = R$ in state $(1,3)$, so the users are accessing tracks in different groups.

In state $(2,3)$, the writer writes on track $d_w = 1 - D_W$ in group $w = W$, while the reader reads from track d in group R. The diagram shows that either $W \neq R$ or $d = D_R$, so the users are again accessing different tracks. □.

6.1 Correctness

Given lemma 3, it remains to show that for every run $(\mathcal{A}, \rightarrow)$, there exists a reading function π such that $\sigma = (\mathcal{A}, \rightarrow, \pi)$ satisfies the three atomicity conditions. As before we may assume that the set of all atomic bit accesses is totally ordered by $\rightarrow$, hence we can use the state model.

Lemma 2 allows us to define the reading function π as the "union" of the four reading functions that make each track atomic. This means that a read is mapped to the write which was the last to access the track from which the read obtained its value. We now prove each of the three conditions in turn.

Proof of A0 The reading function is obviously normal by the safety of the track-bits.

Proof of A2 The proof is by contradiction. Let $r \in \mathcal{R}, w \in \mathcal{W}$ be such that $\pi(r) \rightarrow w \rightarrow r$.[4] Assume without loss of generality that w writes on track $T_{0,0}$ and that $D_1 = 0$ at time $f(w)$. Then at the same time, $W = 0$ and $D_0 = 0$.

Consider now the 4 possible tracks that r can read from:

$T_{0,0}$ This contradicts the assumption that $\pi(r)$ precedes w.

$T_{0,1}$ In this case, r reads $d = 1$ from D_0, which requires that the writer changes D_0 to 1 between $f(w)$ and the read of D_0 by r. But according to the writer protocol, this change is preceded by the writing of track $T_{0,1}$, implying $w \rightarrow \pi(r)$ and hence leading to a contradiction. The last two cases are similar and we need only show that the track read by r must have been written after w.

$T_{1,0}$ In this case, r reads 1 from W, which requires that the writer changes W to 1. This is preceded by the writing of track $T_{1,0}$.

$T_{1,1}$ In this case, r reads $d = 1$ from D_1, which requires that the writer changes D_1 to 1. This is preceded by the writing of track $T_{1,1}$.

In all three cases, we see that r cannot read a value older than that of w, because the display (W, D_0, D_1) doesn't change until the new track has been written. In other words, once the display is set, every new read action must read either the track on display or a more recently written one.

Proof of A1 We claim that A1 follows from A2 and show this by deriving a violation of A2 from a violation of A1. Let $r_1, r_2 \in \mathcal{R}$ be such that $r_1 \rightarrow r_2$ and $\pi(r_2) \rightarrow \pi(r_1)$. By definition of π and lemma 3, r_1 accesses some track, say $T_{0,0}$, after $\pi(r_1)$ does so. But since $\pi(r_1)$ ends its track access by changing an atomic bit (W or D_0) and thereby finishing its protocol, we have that $f(\pi(r_1)) < f(r_1) < s(r_2)$, hence with $w = \pi(r_1)$, $\pi(r_2) \rightarrow w \rightarrow r_2$, violating A2. □

6.2 Space Complexity

Now that the 4-track construction has been proven correct, we consider its "space complexity." Using the 3 safe bit construction to implement each of the four atomic bits, we see that 12 bits suffice for the switch. But we can do better, because those atomic bits are used in a special way. In particular, since the W and R bits are used for handshaking, there is exactly one atomic read of W between an atomic change of W and an atomic change of R (and vice versa). Hence there is at most one atomic change of W between two consecutive atomic reads of W (and vice versa). With the trackdisplay bits D_0, D_1 the situation is more complicated. When the reader changes groups (say, to 0), and atomically reads D_0, there can be at most one atomic change of D_0 before the writer leaves group 0.

We will show that, because of these properties, we can implement any of the four atomic bits, call it B, with 2 safe bits B_0, B_1. The problem with safe bits is their flickering. If, for example, R was only a safe bit, then while being changed by the reader, the writer could first see the new value, change groups, then see the old value and write to the displayed track in the old group. With 2 safe bits, the following scheme can be applied to alleviate the flickering problem. We represent the value of atomic bit B as the exclusive-or (xor,$\oplus$) of 2 safe bit values: $B = B_0 \oplus B_1$. The change of atomic bit B is then replaced by a change of safe bit B_b, where b is the old value of B. Thus, B_0 and B_1 are changed alternatingly. For the purpose of reading B, two local copies b_0, b_1 of B_0 and B_1 are kept. Normally then, an atomic read of B is replaced by a safe read of B_b, where $b = b_0 \oplus b_1$ is the old value of B. In this case, new-old inversions are eliminated, since the flickering bit is no longer examined once the new value is obtained. This procedure suffices for reading W and R, since the handshaking ensures that each safe bit change is noticed by the other user. It also suffices if

[4] It will be clear from context whether we mean the write action w or the similarly denoted writer's local copy of W.

the reader sees the writer in the same group and wants to read the trackdisplay, because the writer will change the display at most once (before moving to the other group). If on the other hand the reader sees the writer in the other group, then any local copies it would have of the trackdisplay bits in that other group would probably be out of date. In this case it can simply read both safe bits of that display one after the other, because again the writer will change the display at most once before moving to the other group.

The new architecture of the switch is as follows:

The corresponding protocols are:

```
WRITER PROTOCOL                    READER PROTOCOL

1.  if R[1-w]==W[1-w] then         if W[r]<>R[r] then
2.    w := 1-w                       change R[r]
3.    write track T[w,x[w]]          r := 1-r
4.    change W[1-w]                  read d[0] := D[r,0]
5.  else                             read d[1] := D[r,1]
6.    x[w] := 1-x[w]               else
7.    write track T[w,x[w]]          read d[d[0]⊕d[1]] := D[r,d[0]⊕d[1]]
8.    change D[w,1-x[w]]           endif
9.  endif                          read track T[r,d[0]⊕d[1]]
```

The writer's local variables d_0, d_1 have been renamed to x_0, x_1 to emphasize that x_i now represents the eXclusive-or of $D_{i,0}$ and $D_{i,1}$. The reader's local variable d has been replaced by d_0 and d_1, where d_i is meant to hold a copy of $D_{r,i}$. All shared and local variables are initialized to 0 as usual. Because the switch now consists of eight safe bits, we call it the "Safe Byte Switch."

We can now state the main theorem:

Theorem 2 *A single-reader, single-writer b-bits atomic variable can be constructed from* $4b+8$ *safe bits (4 tracks and a safe byte).*

We postpone the proof of correctness of the new construction to section 7.3.

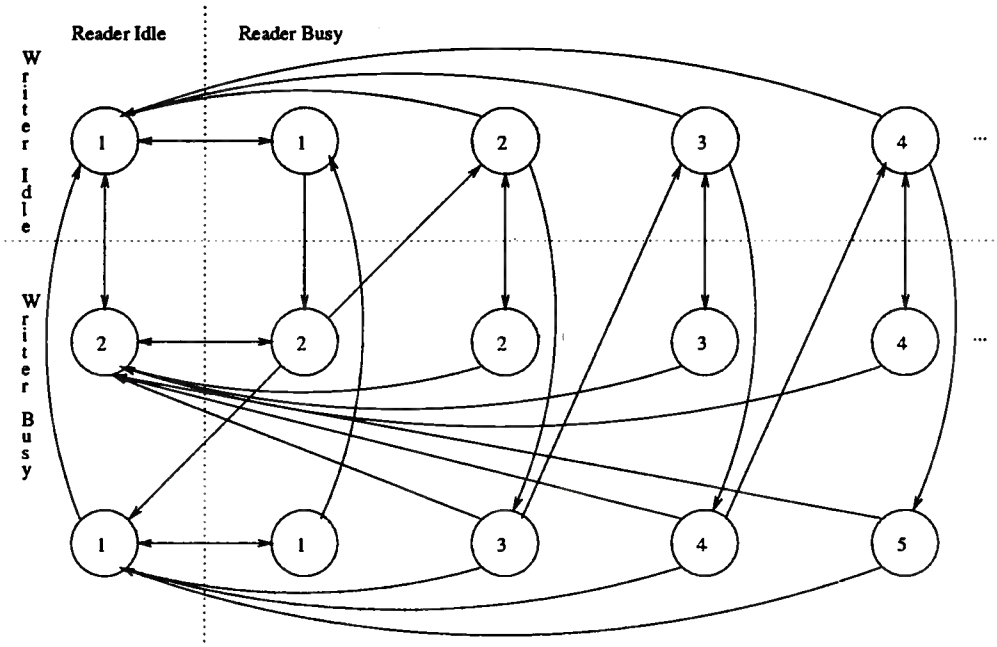

Figure 2: the general atomicity automaton

7 The Atomicity Automaton

In this and the next few sections we discuss the use of machines (computers) as an aid in designing and verifying atomic variable constructions.

The verification is based on a generic automaton which embodies the three atomicity properties of system executions (in the single-reader, single-writer case). Figure 2 shows a picture of the automaton. The transitions of this automaton represent the starts and ends of read and write actions, while the nodes represent the "atomicity state" of a run on the atomic variable. The latter corresponds to the set of values that the next-ending read can return without violating atomicity—its size is shown inside each node.

The nodes can be divided in four groups, depending on whether each user is idle or busy accessing the variable. When both users are idle, the atomicity state of the run is fixed by the current value of the variable—this being the only value that a newly started read is allowed to return. This explains the single node in this group.

When the writer is busy and the reader idle, we can distinguish between two states: either the reader has read the new value that is being written, or it hasn't. Hence there are two nodes in this group. In the former case subsequent read actions must return the same value as the last read action in order to prevent new-old inversions (condition A1). Hence the set size of one.

When the reader is busy, there are many possible states, depending on the number of writes that overlap the read. As the nodes progress to the right, the set of values that the current read action is allowed to return, grows. Of course, while the picture suggests an infinite progression of nodes, its size is in fact limited by the number of values that the atomic variable can hold (its domain-size).

In the group where only the reader is busy, there are two start-of-write transitions vertically emanating from each node. As can be deduced from the resulting set sizes, the upper transition corresponds to the write of a value already in the set of permitted return values. In this case, while the set size remains the same, it is now no longer required to map the read action to the current write action (if the read action decides to return its value). This means that the next read action will not be able to combine with the current one to create a new-old inversion.

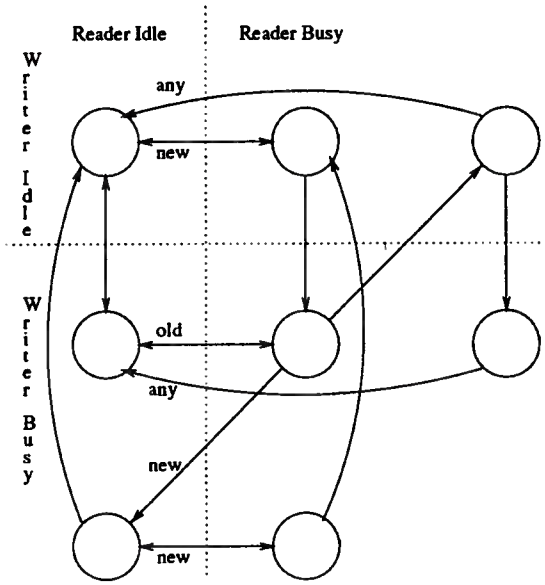

Figure 3: the atomic bit automaton

Alternatively, if the value of the new write is outside the set, then this value is added to it, but if now the read decides to return the new value, then the atomicity state represented by the bottom-left node is reached. The other leftward transitions from the right-bottom nodes to the left middle node correspond to return values written by earlier write actions.

7.1 Using the Automaton for Verification of a given Run

For verification, we include in the state information the actual set of values of *completed write actions* that are valid return values for the next ending read. The value returned by a read can then be verified as follows: If it is in the above set, then we take the leftward transition to the top-left or the middle-left node, and reduce the set to contain only the value of the last completed write. Otherwise, if it is the value currently being written by the writer, then we take the leftward transition to the bottom-left node and empty the set. If the returned value satisfies neither of these cases, then the run is non-atomic. The set is further maintained at the completion of a write, by either adding the written value to the set if the reader is busy, or changing the set to the singleton with that value if the reader is idle.

For atomic bit constructions, we know that the values written are alternatingly 1 and 0. This means that the size of the set of permitted return values is bounded by 2. The size of the atomicity automaton shrinks acccordingly. In the group with only the reader busy, there are only two essentially different nodes—either the reader can return the current value of the atomic bit, or it can return both 0 and 1. In the group where both users are busy, there are only three nodes. In one, it can return either the old or the new value with two different transitions. In the second, it can return both 0 and 1 as old values so there is only one such transition. In the third node, it must return the new value. Figure 3 shows this reduced automaton, with explicit mention of which value is returned by a read (if the writer is idle, then "new" means "current.") There are two nodes from which a single new-labelled transition emanates to the left. From these nodes atomicity can be violated if the read returns the other value.

7.2 Verifying the Atomic Bit Construction

A program has been written to systematically search all states of the atomic bit construction. The state information involves the following:

- position of writer in its protocol, i.e., writer state
- position of reader in its protocol, i.e., reader state
- values of the reader's local variables
- values of the three safe bits
- position in automaton, i.e., atomicity state

We now explain how the safety of the shared bits is modelled. A safe write is modelled by two separate transitions representing the start and the finish of that write. A read, on the other hand, is represented by a single transition, as if it occurred in a single time instant. This can be done for the following reason. If a read from a safe bit overlaps a write on the same bit, then either 0 or 1 can be returned, so the read might as well have occurred completely within that write. If no write overlaps the read, then the value returned must be that of the last preceding write, and it clearly doesn't matter how long the read lasts.

In summary, if a read occurs between two consecutive writes, then there is a single transition corresponding to the return of the current value, and if it occurs between the start and finish of a write, then there are two different transitions, one for each value that can be returned.

This model captures the essence of safe bits. It leads to 3 writer states and 7 reader states. The program starts by putting the initial state in an otherwise empty set. Then it repeatedly takes an element from the set, and replaces it by all states that result from the removed one by a single transition and are not yet in the set. Additionally, the program keeps track of the shortest path from the initial state to each visited one. If some transition is the return of a value which is invalid according to the automaton, then the program prints out a description of the shortest path to the failing state, revealing the shortcomings of the construction being verified. Otherwise, if the set becomes empty, then some statistics are printed such as the number of visited states for each combination of writer state and reader state.

The program proved to be of great help during the design of the atomic bit construction, making it easy to try out various alternatives, and immediately getting a "diagnosis" of possible problems.

7.3 Verifying the Safe Byte Switch Construction

Like in the proof of the 4-track construction with the 4 atomic bits, we must first establish that the new construction is collision free, that is, we must prove lemma 3 again. For this we again need an invariant to hold under all possible runs based on a state diagram. In this case however, we cannot assume that the switch bit accesses can be linearized, since they are only safe. Instead we adopt the safe bit model of the previous section. This entails redefining the reader and writer states. Experimentation has shown that the simplicity of the invariants depends rather heavily on the exact form of the protocols. The following protocols, semantically equivalent to those of section 6.2 (local assignments have moved and indices changed accordingly), proved to give the best results:

```
WRITER PROTOCOL                    READER PROTOCOL

1.  if R[1-w]==W[1-w] then         if W[r]<>R[r] then
2.    write track T[1-w,x[1-w]]      r := 1-r
3.    w := 1-w                       change R[1-r]
4.    change W[1-w]                  read d[0] := D[r,0]
5.  else                             read d[1] := D[r,1]
6.    write track T[w,1-x[w]]     else
```

```
7.    x[w] := 1-x[w]                read d[d[0]⊕d[1]] := D[r,d[0]⊕d[1]]
8.    change D[w,1-x[w]]          endif
9.  endif                         read track T[r,d[0]⊕d[1]]
```

Note that when a safe bit is changed, the local copy already holds the new value—this is the property that ensures the most simple invariants. We proceed to enumerate the essential positions of the users in their protocols.

The states of the writer are:

0 idle, i.e., before the safe read of R_{1-w} in line 1

1 between the safe read of R_{1-w} and the safe change of W_{1-w} in line 4, when it is writing track $T_{1-w,x_{1-w}}$

2 changing safe bit W_{1-w} in line 4

3 between the safe read of R_{1-w} and the safe change of $D_{w,1-x_w}$ in line 8, when it is writing track $T_{w,1-x_w}$

4 changing safe bit $D_{w,1-x_w}$ in line 8

Thus, the writer is always moving from state 0 to either state 1 followed by state 2 or to state 3 followed by state 4 (depending on the outcome of the test), and then back to state 0.

The states of the reader are:

0 idle, i.e., before the safe read of W_r in line 1

1 between the safe read of W_r and the safe read of $D_{r,0}$ in line 4, when it is changing safe bit R_{1-r}

2 between the safe read of $D_{r,0}$ and the safe read of $D_{r,1}$ in line 5

3 between the safe read of W_r and the safe read of $D_{r,d_0\oplus d_1}$ in line 7

4 reading track $T_{r,d_0\oplus d_1}$ in line 9

Thus the reader is always moving from state 0 to either state 1, followed by state 2 or to state 3, then on to state 4, and finally back to state 0.

Altogether, there are now $5 \times 5 = 25$ states, which are pictured in figure 4, along with all possible transitions. Each node contains a (possibly empty) set of formulas, which are to be conjuncted, together with the invariant $W_w = R_w$ which holds for all nodes. To solve the potential ambiguity of this formula which arises when the reader is changing R_w (the writer changes W_{1-w}), we make the following definition:

Definition 5 *If a safe bit B is being changed then in formulas, B refers to its* new *value.*

The notation $D(i)$ is used as an abbreviation of $d_i = D_{r,i}$ and expresses that a local display bit of the reader matches the shared one (in the reader's group). By lack of an arrow symbol, a greater-than sign ($>$) is used to denote implication. Starting from the initial state $(0,0)$ (both users idle), each invariant can be manually checked by considering the possible predecessors of a node.

Another state space search program was used to construct this diagram. It's state also includes information on the validity of the tracks (relative to the next end-of-read), as derived from the atomicity automaton. With the help of this information, the program actually verifies the correctness of the construction. In this paper however, we neglect this extra state, since including it would make the diagram overly complex, making manual inspection practically impossible.

It is now easy to see from the diagram that lemma 3 holds. Potential collisions can occur in states $(1,4)$ and $(3,4)$. In the first case, $w = r$ and the writer writes on a track in group $1-w$, so there is no collision. In the second case, the writer writes on track $T_{w,1-x_w}$. If $w \neq r$, then we are done. Otherwise, the reader and writer are in the same group, so $w = r$, implying $d_{1-x_r} = D_{r,1-x_r}$. Along with $d_{1-x_r} = D_{r,1-x_r}$, this leads to $x_r = D_{r,0} \oplus D_{r,1} = d_0 \oplus d_1$, so the reader is on the other track—no collision.

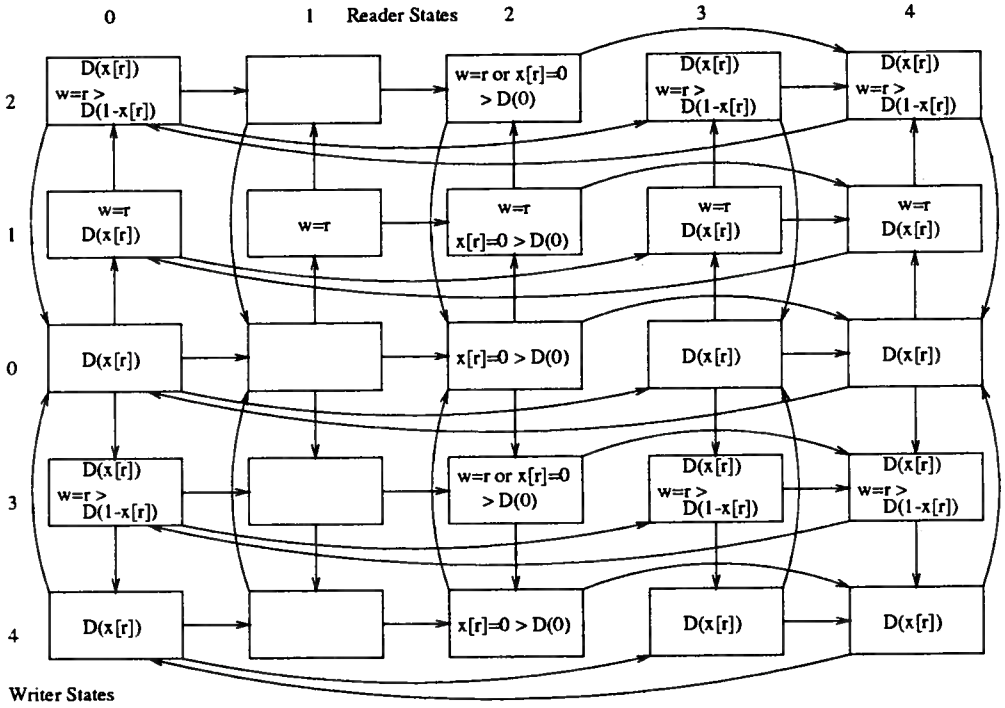

Figure 4: state diagram of 4-track construction with safe byte switch

8 Correctness of Safe Byte Switch Construction

Given lemma 3, it remains to show that for every run $(\mathcal{A}, \rightarrow)$, there exists a reading function π such that $\sigma = (\mathcal{A}, \rightarrow, \pi)$ satisfies the three atomicity conditions. We choose the reading function π to map a read action to the write action that last writes to the track before the read action reads from that track. As before, this can be viewed as the union of the four reading functions that make each track atomic, according to lemmas 3 and 2.

We now prove each of the three conditions in turn.

Proof of A0 The reading function is obviously normal by the safety of the track-bits.

Proof of A1 The proof is by contradiction. Let $r_1, r_2 \in \mathcal{R}$ be such that $r_1 \rightarrow r_2$ are two consecutive read actions and $\pi(r_2) \rightarrow \pi(r_1)$. Assume that $\pi(r_1)$ writes on track $T_{0,0}$, that $W_0 = W_1 = R_0 = 0$ (using the invariant $W_w = R_w$) and that $D_{1,0} = D_{1,1} = 0$ at time $f(\pi(r_1))$.[5]

Consider now the 4 possible tracks that r_2 can read from:

$T_{0,0}$ This contradicts the assumption that $\pi(r_2)$ precedes $\pi(r_1)$, which cannot be the case if r_1 and r_2 read from the same track.

$T_{0,1}$ Note that r_2 didn't change groups thus taking the else branch. Examination of figure 4 reveals that $d_0 = D_{0,0}$ at time $f(r_1)$ (Recall that $x_i = D_{i,0} \oplus D_{i,1}$). In order for r_2 to read track $T_{0,1}$, it must have seen a change in $D_{0,0}$, which it reads in line 7. But $\pi(r_1)$ ends by changing either W_1 (line 4) or $D_{0,1}$ (line 8). Hence a write action later than $\pi(r_1)$ started changing $D_{0,0}$ before r_2 accessed its track, and this write action must have written to that track, contradicting $\pi(r_2) \rightarrow \pi(r_1)$.

$T_{1,0}$ In this case r_2 did change groups, taking the then branch. So in line 1, it saw W_0 set ($W_0 \neq R_0 = 0$). Given that $\pi(r_1)$ doesn't change W_0, a later write action must have started changing it, following the writing on track $T_{1,0}$. This again contradicts $\pi(r_2) \rightarrow \pi(r_1)$.

$T_{1,1}$ Again r_2 changed groups and took the then branch. Also, it saw either $D_{1,0}$ or $D_{1,1}$ set, which requires that a write action later than $\pi(r_1)$ has already scribbled on track $T_{1,1}$, contradicting $\pi(r_2) \rightarrow \pi(r_1)$.

Proof of A2 The proof is once again by contradiction. Let $r \in \mathcal{R}, w \in \mathcal{W}$ be such that $\pi(r) \rightarrow w \rightarrow r$. Assume that w writes on track $T_{0,0}$, that $W_0 = W_1 = R_0 = 0$ and that $D_{1,0} = D_{1,1} = 0$ at time $f(w)$.

Consider now the 4 possible tracks that r can read from:

$T_{0,0}$ This contradicts our choice of the reading function π, since $\pi(r)$ must either equal w or succeed it.

$T_{0,1}$ Since $\pi(r) \rightarrow w$, track $T_{0,1}$ is not written to between w and its access by r. Study of the writer protocol then shows that $D_{0,0}$ and $D_{0,1}$ remain constant during that time. Because w and r access different tracks, r did not read both $D_{0,0}$ and $D_{0,1}$—it took the else branch. From figure 4 we obtain that, with r in state 3, $d_{x_r} = D_{r,x_r}$. But then either d_{1-x_r} already equals $D_{r,1-x_r}$, or it will do so after the read in line 7, in contradiction with $\pi(r) \neq w$.

$T_{1,0}$ Since $\pi(r) \rightarrow w$, track $T_{1,0}$ is not written to between w and its access by r. Study of the writer protocol now shows that W_0 and W_1 remain 0 during that time. But then in line 1, r reads either W_1 (and moves to group 0), or W_0, in which case it remains in group 0, a contradiction.

$T_{1,1}$ Since $\pi(r) \rightarrow w$, track $T_{1,1}$ is not written to between w and its access by r. Study of the writer protocol now shows that $D_{1,0}$ and $D_{1,1}$ remain 0 during that time. We conclude that r takes the else branch. Again with r in state 3, we have $d_0 = d_{x_r} = D_{r,x_r} = 0$. But then either already $d_1 = 0$, or this will hold after the read in line 7, in contradiction with $\pi(r) \neq w$.

□

[5] Other cases are analogous.

9 Conclusions

We have presented and proven correct the following two constructions:

- an atomic bit from 3 safe bits
- an atomic b-bit variable from $4b + 8$ safe bits

The first achieves the optimal number of non-atomic bits needed (optimal space complexity). The second needs only 2 extra bit accesses in a write action, and at most 4 extra bit accesses in a read action on the atomic variable (in addition to the b accesses to the bits on a track), making its time complexity very near (if not equal) to optimal. The cost for this "speed" is in the space complexity, which is about a factor 4/3 from optimal, since Peterson showed the sufficiency of 3 tracks. A main advantage of the 4-track construction as given here, is its simplicity and transparency—the purpose of the bits in the architecture and the workings of the protocols can be easily understood. We have developed a finite state verification methodology for concurrent wait-free shared variable constructions, whose succesful application provides additional practical support.

10 Acknowledgements

I want to thank P.M.B. Vitányi for discussions, useful suggestions and his careful proofreading.

References

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

[2] E.M. Clarke, E.A.Emerson, A.P. Sistla, *Automatic Verification of Finite State Concurrent Systems Using Temporal Logic Specifications: A Practical Approach*, Proc. 10th ACM Symposium on Principles of Programming Languages, pp. 117–126, 1983.

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

[4] L.M. Kirousis, E. Kranakis, P.M.B. Vitányi, *Atomic Multireader Register*, Proc. 2nd International Workshop on Distributed Computing, Springer Verlag Lecture Notes in Computer Science 312, pp. 278–296, 1987.

[5] M. Li, J. Tromp, P.M.B. Vitányi, *How to Share Concurrent Wait-Free Variables*, CWI Technical Report CS-R8916, April 1989.

[6] B. Awerbuch, L.M. Kirousis, E. Kranakis, P.M.B. Vitányi, *On Proving Register Atomicity*, Proc. 8th Conference on Foundations of Software Technology and Theoretical Computer Science, Springer Verlag Lecture Notes in Computer Science 338, pp. 286–303, 1988.

[7] 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.

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

[9] J.E. Burns, G.L. Peterson, *Sharp Bounds for Concurrent Reading While Writing*, Technical Report, Georgia Institute of Technology GIT-ICS-87/31

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

[11] 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)