A Randomized Algorithm for Two-Process Wait-Free Test-and-Set

John Tromp
CWI*

Paul Vitányi
CWI & UvA†

Abstract

The work of Herlihy and Plotkin on universal shared objects shows how to implement arbitrary wait-free read-modify-write objects from a bounded number of shared read/write bits. deterministically in a concurrent setting using only atomic shared variables. We present a simple randomized algorithm for 2-process wait-free test-and-set, implemented with two 4-valued single writer single reader atomic variables. The worst-case (over all adversarial schedulers) expected number of steps to execute a test-and-set between two processes is at most 11, while a reset takes exactly 1 step. Based on a finite-state analysis, the proofs of correctness and expected length are compressed into one table.

1 Introduction

A concurrent system consists of n processes communicating through concurrent data objects $R_0, \ldots, R_{m-1}$. An implementation of a new concurrent data object x is *wait-free* if there is a total function f, such that each process can complete any operation associated with x within $f(n)$ steps, irrespective of the timing and execution speeds of the other processes. A step is a single access to one of the R_i's. Local events, including coin-flips, are not counted.

*Centrum voor Wiskunde en Informatica, Kruislaan 413, 1098 SJ Amsterdam, The Netherlands. email: tromp@cwi.nl

†Faculteit Wiskunde en Informatica, Universiteit van Amsterdam. email: paulv@cwi.nl

The paper develops a similar definition of wait-freedom for randomized protocols based on the worst case expected length of an operation.

A concurrent object is *constructible* if it can be implemented deterministically with boundedly many safe bits, the mathematical analogues of electronic hardware 'flip-flops', [11]. What concurrent wait-free object is the most powerful constructible one? It has been shown that wait-free atomic multi-user variables, and atomic snapshot objects, are constructible [16, 11, 21, 10, 18, 13, 7, 4, 2]. In contrast, wait-free consensus—viewed as an object on which each of n processes can execute just one operation—is not constructible [6, 1], although randomized implementations are possible [6, 1, 5, 19]. Wait-free concurrent test-and-set can deterministically implement 2-process wait-free consensus, and therefore is not deterministically constructible [12, 8]. This raises the question of whether randomized algorithms for test-and-set exist.

In [8] it is shown that repeated use of 'consensus' on *unbounded* hardware can implement 'test-and-set'. In [17, 19, 9] a solution can be obtained by combining several intermediate constructions.

Wait-free n-process test-and-set can be implemented deterministically from wait-free 2-process test-and-set, [3], showing that the impossibility of a deterministic algorithm for n-process test-and-set is solely due to the 2-process case. We present a simple randomized algorithm which directly implements wait-free test-and-set between 2 processes. Randomization means that the algorithm contains a branch conditioned on the outcome of a fair coin flip (as in [20]). We use a finite-state based proof technique for verifying correctness and worst-case expected execution length.

The solution is very simple: it uses two 4-valued 1-writer 1-reader atomic variables. The worst-case expected number of steps in a test-and-set operation is 11, whereas a reset always takes 1 step.

2 Preliminaries

A *test-and-set* bit is a concurrent data object X shared between n processes $0, \ldots, n-1$. The value of X is 0 or 1. Each process i has a local binary variable x_i. At any time exactly one of $X, x_0, \ldots, x_{n-1}$ has value 0, all others have value 1. A process i with $x_i = 1$ can atomically execute a test-and-set operation read $x_i := X$; write $X := 1$; return x_i.

A process i with $x_i = 0$ can execute a reset operation

$x_i := 1$; write $X := 0$.

This naturally leads to the definition of the *state* of the test-and-set bit, or *0-owner* as a member of $\{\perp, 0, \ldots, n-1\}$ according to which of X and the x_i's is 0.

2.1 Test-and-Set Definition

Instead of assuming an atomic test-and-set, we want to implement it with actions that are sequences of accesses to atomic shared variables, $R_0, \ldots, R_{m-1}$, executed by processes $0, \ldots, n-1$, according to some *protocol*. Since the executions by the different processes happen concurrently and asynchronously, the implementation should guarantee that each system execution of the implementation is equivalent to a system execution of the above defined construct. This leads to the following set of definitions.

Definition. For a given execution of the system, denote the set of *actions* that have been started as $A = R \cup T, T = T0 \cup T1$, where R is the set of *resets*, and Tx is the set of *test-and-sets* returning x, $x = 0, 1$. By $r, t, t0, t1$ we denote elements from $R, T, T0, T1$, respectively. We partition the set of actions according to the processes executing them: define A_i to be the actions by process i, and similarly define $R_i = A_i \cap R$, and $Tx_i = A_i \cap Tx$, $x = \epsilon, 0, 1$. Define an *event* as an execution of a statement in a protocol, that is, a write or a read on a R_i or a coin-flip. A read event is qualified by the value obtained and a coin-flip by its outcome. Every new execution of a statement represents a unique event. Number the events of a test-and-set or reset action $a \in A$ as $a.1, a.2, \ldots$. Let $l = l(a)$, the *length* of a, be its number of events in the execution (possibly infinite). Denote $a.1$, the *start* of a, as $s(a)$. If a finishes during the execution, then denote $a.l$, the *finish* of a, as $f(a)$. Each event is assumed to execute *atomically.* The sequence of the events of all actions in A is called the *history*. A history induces a partial ordering of the actions: $a \to b$ iff $f(a) < s(b)$ (the last event of a precedes the first of b in the history). The number of b such that $b \to a$ is assumed to be finite for each a.

The pair $(A, \rightarrow)$ is called a *run*. An implementation of a concurrent object shared between processes $0, \ldots, n-1$, such that each run $(A, \rightarrow)$ satisfies the following atomicity axiom, is an atomic test-and-set.

Atomicity: We can extend $\rightarrow$ on A to a total order $\Rightarrow$ on A in which the sequence of actions satisfies the test-and-set semantics:

1. the system is initially in state $\perp$.
2. from state $\perp$, an action $t0 \in T0_i$ moves the system to state i.
3. from state i, an action $r \in R_i$ moves the system to state $\perp$.
4. from state i, an action $t1 \in T1 - T1_i$ leaves the system in state i.
5. no other state transitions than the above are allowed.

2.2 Randomization, Adversaries and Wait-Freeness

In the above definition of atomicity we did not use the notion of an adversary. The reason is that atomicity must hold for all possible histories, and hence for all possible outcomes of coin-flips. The adversary is introduced to enable a quantification of the *wait-freedom*. While it is inevitable that for some histories a test-and-set action may last arbitrarily many steps, the probability of such histories occurring should be minimized. This leads us to define the probability of a certain history occurring.

Fix a protocol P. Let H (H^∞) be the set of finite (infinite) histories that can arise from this protocol. I.e. the set of h such that

1. for all $i < n$, $h|A_i$, the restriction of h to events by process i, satisfies the protocol for process i, and
2. for all $j < m$, $h|R_j$, the restriction of h to events that access R_j, satisfies the usual semantics of such an atomic variable (a read event returns the value written by the last write event preceding it).

For $h \in H$, let the *cylinder* Γ_h be the set of all histories in H^∞ that start with h. Write he to denote history h followed by event e.

An *adversary* is then defined as a probability measure μ on H^∞ satisfying:

1. $\mu(\Gamma_\epsilon) = 1$, where ϵ is the empty history;

2. $\mu(\Gamma_h) = \sum_{h\,e \in H} \mu(\Gamma_{h\,e})$, for $h \in H$ and e is a single event; and

3. $\mu(\Gamma_{h\,c(\text{heads})}) = \mu(\Gamma_{h\,c(\text{tails})})$, for each coin-flip event $c()$ with $h\,c() \in H$.

The first two conditions—already implied by the notion of probability measure—are included for completeness. The third condition ensures that the adversary has no control over the outcome of a fair coin flip: both outcomes are equally likely. This definition is readily generalized to biased coins and multi-branch decisions.

Note that this notion of adversary is the strongest possible short of allowing it to predict the future. For example, it includes nonrecursive adversaries using omniscient oracles and randomization.

Now that adversaries have been defined, we can define the expected length $E(h,i)$ of process i's current (or next, if idle) action following a finite initial history segment h. Let $\omega \in \Gamma_h$ be an infinite history starting with h. Let $l_{h,i}(\omega)$ be the length (number of events) of process i's current action following h in ω. If process i is idle at h, then by 'current' we mean 'next', leaving $l_{h,i}$ undefined for ω in which a doesn't start a new action. Define

$$E(h,i) = \sum_{k=1}^{k=\infty} k \cdot \frac{\mu(\{\omega \in \Gamma_h : l_{h,i}(\omega) = k\})}{\mu(\Gamma_h)}.$$

The summation includes the case $k = \infty$ so that the expected length is infinite if (but not necessarily only if) the set of infinite histories in which an operation execution has infinitely many events, has positive measure. The normalization w.r.t. h gives the adversary a free choice of 'starting' configuration.

Definition. An implementation of a concurrent object shared between n processes is *wait-free*, if there is a constant $f(n)$ bounding the expected length $E(h,i)$, for all h, i, under all adversaries.

3 Solution for Two Processes

We give a test-and-set implementation between two processes, process 0 and process 1. The construction uses two 4-valued shared variable objects, R_0 and R_1. The four values are 'me', 'he', 'choose', 'rst'. Process i solely writes

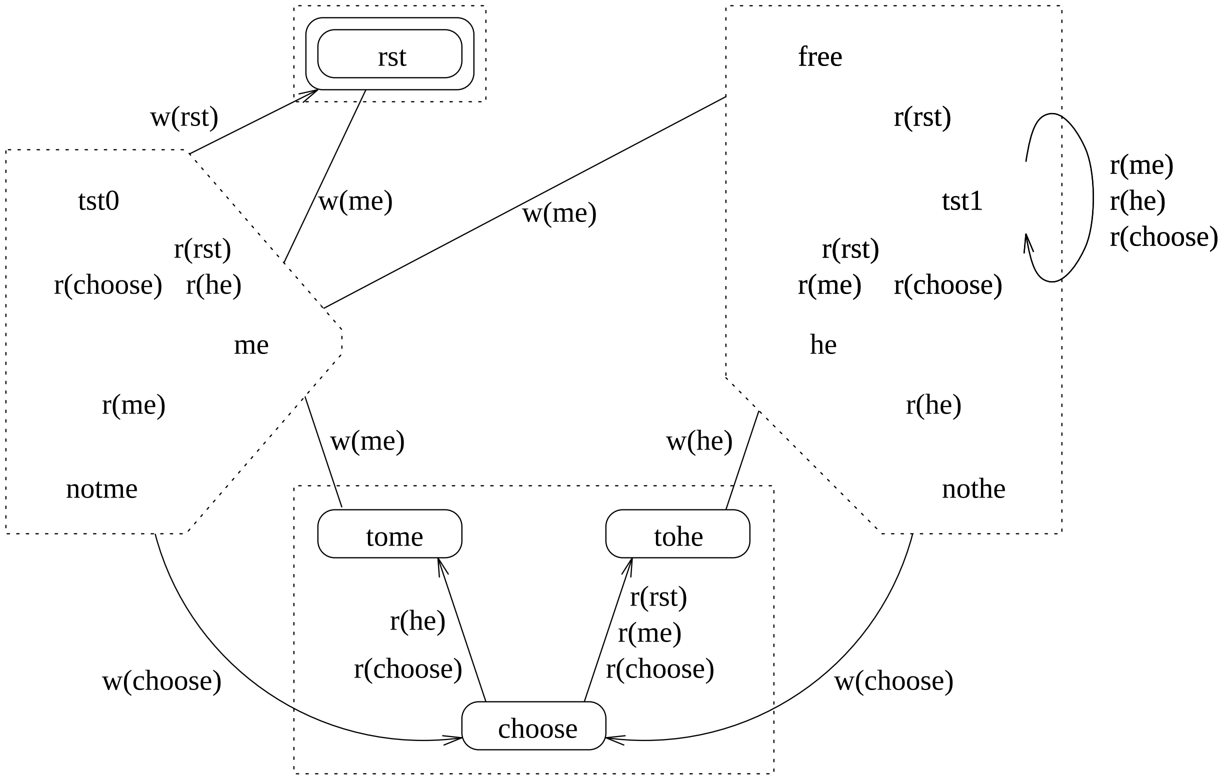

Figure 1: test-and-set protocol

variable R_i, its *own* variable, and solely reads R_{1-i}. For this reason the reads and writes in the protocol don't need to be qualified by the shared variables they access. The protocol, for process i, is first presented as a finite state transition diagram, figure 1. The transitions are labeled with reads $r(value)$ and writes $w(value)$ of the shared variables, where $value$ denotes the value read or written. The 11 states of the protocol are split into 4 groups enclosed by dotted lines. Each group is an equivalence class consisting of the set of states in which that process's variable has the same value. That is, the states in a group are equivalent in the sense that process $1-i$ cannot distinguish between them by reading R_i. Accordingly, the inter-group events are writes to R_i, whereas the intra-group events are reads of R_{1-i}. Each group is named after the corresponding value of the shared variable. The diagram is deterministic, but for a coin flip which is modeled by the two $r(choose)$ transitions from the choose state.

A more conventional representation of the protocol, for process i, is given below. An occurrence of R_i not preceded by 'write' (resp. R_{1-i} not preceded by 'read') refers to the last value written to it (resp. read from it), stored in correspondingly named local variables. The conditional 'rnd(true,false)' represents the boolean outcome 'true' or 'false' of a fair coin flip. The system is initialized with all local and global variables in state rst.

```
test_and_set:

if R_i = he AND read R_{1-i} ≠ rst
then return 1
write R_i := me
while read R_{1-i} = R_i do
  write R_i := choose
  if read R_{1-i} = he OR (R_{1-i} = choose AND rnd(true,false))
  then write R_i := me
  else write R_i := he
if R_i = me
then return 0
else return 1

reset:
```

`write` $R_i :=$ `rst`

It can be verified in the usual way that the transition diagram represents the operation of the program. The intuition behind the protocol is as follows. The default situation is where both processes are idle in the rst state. If process i starts a test-and-set then it writes $R_i :=$ me (indicating its desire to take the 0), and checks whether process $1-i$ agrees (by *not* having $R_{1-i} :=$ me). If so, then it has successfully completed a test-and-set of 0. It is easy to see that in this case process $1-i$ can not get 0 until process i does a reset by writing $R_i :=$ rst. While $R_i =$ me, process $1-i$ can only move from state 'me' to state 'notme' and on via states 'choose', 'tohe' and 'he' to 'tst1', where it completes a test-and-set of 1.

Problems arise only if both processes see each other's variable equal to 'me'. In this case they are said to *disagree* or *in conflict*. They then proceed to the choose state from where they decide between going for 0 or 1, according to what the other process is seen to be doing. (It is essential that this decision be made in a neutral state, i.e. without a claim of preference for either 0 or 1. If, for example, on seeing a conflict, a process would change preference at random, then a process cannot know for sure whether the other one agrees or is about to write a changed preference.)

The deterministic choices, those made if the other's variable reads different from 'choose', can be seen to lead to a correct resolution of the conflict. A process ending up in the tst1 state makes sure that its test-and-set of 1 is justified, by remaining in that state until it can be sure that the other process has taken the 0. Only if the other process is seen to be in the rst state need it try to take the 0 itself.

Suppose now that process i has read $R_{1-i} =$ choose and is about to flip a coin. Assume that process $1-i$ has already moved to one of the states tome/tohe (or else reason with the processes interchanged). With 50 percent chance, process i will move to the opposite state as did process $1-i$, and thus the conflict will be resolved.

So, intuitively, the probability of each loop through the choose state is at most one half and the expected number of 'choices' (transitions from state choose) at most two. This shows that the worst case expected test-and-set length is 11. Namely, starting from the tst1 state, it takes 4 steps to get to state choose, another 4 steps to loop back to choose and 3 more steps to reach tst0/tst1. The reset operation always takes 1 step.

4 Proof of correctness of the 2-Process Solution

Let h be a history corresponding to a run $(A, \rightarrow)$ of our implementation. Let $B = \{s(t), f(t) : t \in T\} \cup \{s(r) = f(r) : r \in R\}$ be the set of events which start or finish an action. Note that $h|B$, the restriction of h to events in B, completely determines the partial order of actions $\rightarrow$. Let $C = \{t* : t \in T\} \cup R$ be the set of atomic occurrences of actions.

The definition of atomic test-and-set for 2 processes, process 0 and process 1, is captured by DFA2, the DFA in figure 2, which accepts all possible sequences of atomic operations (all states final). The states are labeled with the owner of the bit. The arcs representing actions of process 1 are labeled, whereas the non-labeled arcs represent the corresponding actions of process 0.

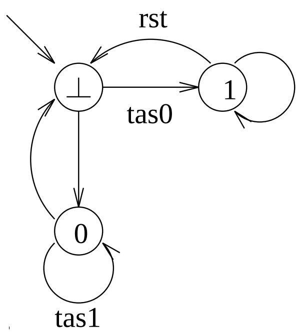

Figure 2: DFA2: specification of 2-process atomic test-and-set

Figure 3 shows the DFA, DFA3, that accepts the possible sequences of the following events of one process (all states final):

- the start of a test-and-set action, denoted s(tas),
- the atomic occurrence of a test-and-set 0, denoted tas0,
- the atomic occurrence of a test-and-set 1, denoted tas1,
- the finish of a test-and-set 0 action, denoted f(tas0),
- the finish of a test-and-set 1 action, denoted f(tas1),

- the reset action, denoted rst.

These are the events in $B \cup C$. The reason for not splitting a reset action into start, atomic occurrence, and finish is that it's implemented in our protocol as a single atomic write where the above three transitions coincide.

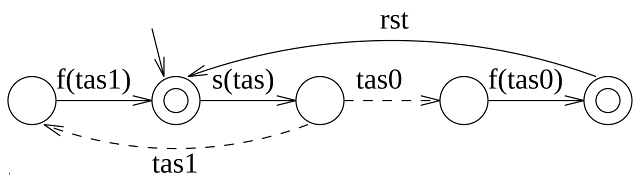

Figure 3: DFA3: non-atomic specification of 1-process test-and-set

The proof is based on the finite state diagram DFA4 in figure 4 below (again all states are final).

It is drawn as a cartesian product of the two component processes—transitions of process 0 are drawn vertically and those of process 1 horizontally. For clarity, the transition names are only given once and only for process 1. Identifying the starts and finishes of test-and-set executions with their atomic occurrences by collapsing the $s()$ and $f()$ arcs, the diagram reduces to the atomic test-and-set diagram. Identifying all nodes in the same column (row) reduces the diagram to the diagram of process 0 (process 1).

In the states labeled 'a' through 'h', neither process owns the 0; the bit is in state $\perp$. In the states labeled 'i' through 'n', process 1 owns the 0; the bit is in state 1. In the states labeled 'o' through 't', process 0 owns the 0; and the bit is in state 0.

Formally [14], DFA4 is the composition of DFA2 with 2 copies of DFA3, in the I/O Automata framework.

Let NFA4 be the NFA obtained from DFA4 by turning the broken transitions of figure 4 into ϵ-steps.

We claim that acceptance of $h|B$ by NFA4 implies atomicity of $(A, \rightarrow)$. This is proven as follows. If NFA4 accepts $h|B$, then, corresponding to the ϵ transitions, we can augment $h|B$ with an atomic transition $t*$ between the start $s(t)$ and finish $f(t)$ of each test-and-set action $t \in T$, to get a history h' accepted by DFA4. Therefore, DFA2, which composes DFA4, accepts $h'|C$,the sequence of atomic events in h'. Furthermore, if $a \rightarrow b$, then $a* \leq f(a) \rightarrow s(b) \leq b*$, so $\Rightarrow$, the total order of actions in $h'|C$, extends $\rightarrow$. This proves atomicity of $(A, \rightarrow)$.

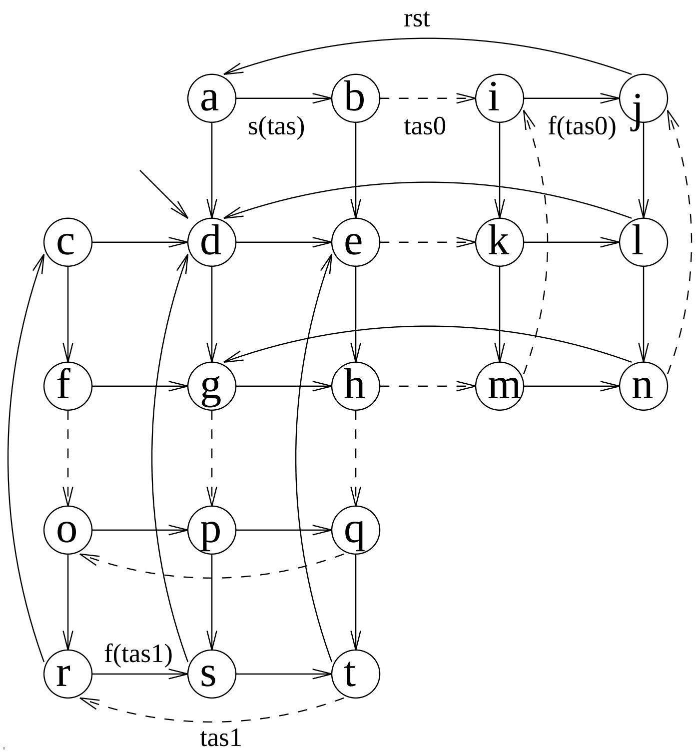

Figure 4: DFA4: non-atomic specification of 2-process test-and-set

To show that for all histories $h \in H$ of our implementation, $h|B$ is accepted by NFA4, and thus the correctness of our construction, we assign to each reachable combination of process states (s_0, s_1) a nonempty set S_{s_0,s_1} of NFA4 states, such that: for each history h ending in process states (s_0, s_1), the set of states in which NFA4 can be after processing $h|B$ contains S_{s_0,s_1} (*). The assignment is given in figure 5. The table entries were chosen so as to minimize the number of ϵ-steps that can be made from each assigned set of NFA4 states. This gives the most insight into the workings of the protocol.

In the table below each row (column) is labeled with a state of process 1 (process 0) as in diagram figure 1. An entry in the table is labeled with roman letters representing a set of atomicity states in figure 4, assigned to that row/column pair of process states. The number ending an entry gives $E(h, 0)$, the expected number of steps to finish the current operation execution of process 0.

We use induction on the length of the history to check (*):

Base: After processing the empty history, NFA4 can be in $\{initialstate\} \supseteq \{d\} = S_{rst,rst}$.

Induction Step: This reduces to checking whether for all transitions (s_0, s_1) to (t_0, t_1) and all NFA4 states $y \in S_{t_0,t_1}$, there is an NFA4 state $x \in S_{s_0,s_1}$, such that NFA4 can move from x to y by processing: either the event corresponding to the transition if it belongs to B, or no event otherwise (there is a sequence of ϵ-steps from x to y).

It is straightforward to check all transitions (state process 0, state process 1) to (newstate process 0, state process 1) or to (state process 0, newstate process 1), corresponding to the atomic transitions in the two copies of protocol figure 1 concerned, to do the induction on the length of the runs to verify correctness, explained above. Simultaneously, the wait-freedom can be checked.

We give an example of checking a few transitions below, and give the interpretation. Verification consists in checking all transitions in the table.

In the default state both processes are in state rst. The table entry $d10$ gives corresponding state d, the start state, in figure 4. The worst-case expected number of steps for a test-and-set by process 0 is 10. Process 0 can start a test-and-set by executing $w(me)$ and entering state me. The corresponding table entry $gp9$ indicates in figure 4 that the system is now

	rst	tst0	notme	me	tome	choose	tohe	he	nothe	tst1	free
rst	d10	l10	cek10	ek10	ek10	c10	c10	c10	c10	d10	ek10
tst0	s1	*	rt1	rt1	rt1	r1	r1	r1	r1	s1	rt1
notme	agp8	jn8	imoq8	imoq8	*	imoq8	imoq8	o4	*	p4	*
me	gp9	jn9	imoq9	imoq9	imoq9	o1	o1	o1	o1	p1	imoq9
tome	gp10	jn10	*	imoq10	imoq10	imoq6	o2	o2	imoq6	p2	*
choose	a3	j3	imoq7	i3	imoq7	imoq7	imoq7	o3	imoq7	p3	*
tohe	a2	j2	imoq6	i2	i2	imoq6	imoq10	imoq10	*	p6	*
he	a1	j1	i1	i1	i1	i1	imoq9	imoq9	imoq9	p5	*
nothe	a4	j4	*	i4	imoq8	imoq8	*	imoq8	imoq8	p4	*
tst1	d11	l11	k11	k11	k11	k11	k11	k11	k11	*	*
free	gp10	jn10	*	imoq10	*	*	*	*	*	*	*

Figure 5: table to verify correctness and wait-freedom

either in state g meaning that process 0 has executed $s(tas)$, or in state p meaning that process 0 has executed $s(tas)$ and also $tas0$ atomically. The expected number of steps is now $9 \leq 10 - 1$. Suppose process 1 now starts a test-and-set: it executes $w(me)$ and moves to state me. The corresponding table entry $imoq9$ gives the system state as one possibility in $\{i, m, o, q\}$ in figure 4 and the expected number of steps for execution of test-and-set by process 0 is still 9. State m says process 1 has executed $s(tas)$ and $tas0$ atomically, while process 0 has only executed $s(tas)$—hence the system was previously in state g and not in state p. State i says process 1 has executed $s(tas)$ and $tas0$ atomically, while process 0 has executed $s(tas)$ and $tas1$ atomically—and hence the system was previously in state g and not state p. States o and q imply the same state of affairs with the roles of process 0 and process 1 interchanged, and the previous system state is either p or g.

Note that it is also consistent for the system to be in state h—neither process having executed tas. However, if both processes have started a test-and-set execution, then necessarily, one of them must return 0. We have optimized the table entries by eliminating such spurious states.

Process 0 might now read $R_1 = me$, and move via state notme (table entry $imoq8$) by writing $R_0 := choose$, to state choose. Process 1 is idle in the meantime. The table entry is now $i3$. This says that process 1 has atomically executed $tst0$, and process 0 has atomically executed $tst1$. Namely, all subsequent schedules lead in 3 steps of process 0 to state tst1—hence the expectation 3.

The expected number of remaining steps of process 0's test-and-set has dropped from 8 to 3 by the last step since 8 was the worst-case which could be forced by the adversary. Namely, from the system in state $(notme, me)$, the adversary can schedule process 1 to move to $(notme, notme)$ with table entry $imoq8$, followed by a move of process 1 to state $(notme, choose)$ with table entry $imoq8$, followed by a move of process 0 to state $(choose, choose)$ with table entry $imoq7$. Suppose the adversary now schedules process 0. It now flips a fair coin to obtain the conditional boolean $rnd(true, false)$. If the outcome is $true$, then the system moves to state $(tome, choose)$ with entry $imoq6$. If the outcome is $false$, then the system moves to state $(tohe, choose)$ with table entry $imoq6$. Given a fair coin, this step of process 0 correctly decrements the expected number of steps. Suppose the adversary schedules process 1 in state $(choose, choose)$. Process 1 flips a fair coin. If the outcome is $true$ the system moves to state $(choose, tome)$ with table entry $imoq7$; if the outcome is $false$ then the system moves to state $(choose, tohe)$ with table entry $imoq7$.

This way the correctness of the implementation can be checked exhaustively by hand. We have done the verification by hand, to optimize the entries, and again by machine.

For the finite-state system as we described, the expected number of remaining steps in a test-and-set execution is *always* bounded by a fixed number. The table shows that, trivially, $1 \leq E(h, 0) \leq 11$ Hence the algorithm is wait-free.

5 On the Difficulty of Multi Process Test And Set

The obvious way to extend the given solution to more than 2 processes would be to arrange them at the leafs of a binary tree. Then, a process wishing to execute an n-process test-and-set, would enter a tournament, as in [15], by executing a separate 2-process test-and-set for each node on the path up to the root. When one of these fails, it would again descend, resetting all the tas-bits on which it succeeded, and return 1. When it succeeds ascending up to the root, it would return 0 and leave the resetting descend to its n-process reset.

The intuition behind this tree approach is that if a process i fails the test-and-set at some node N, then another process j will get to the root successfully and thus justify the value 1 returned by the former.

The worst case expected length of the n-process operations is only $\log n$ time more than that of the 2-process case.

Unfortunately, this straightforward extension does not work. The problem is that the other process j need not be the one responsible for the failure at node N, and might have started its n-process test-and-set only after process i completes its own. Clearly, the resulting history cannot be linearized.

Nonetheless, it turns out that with a somewhat more complicated construction we can *deterministically* implement n-process test-and-set using 2-process test-and-set as primitives, [3]. This shows that the impossibility of deterministic wait-free atomic n-process test-and-set is completely due to the impossibility of deterministic wait-free atomic 2-process test-and-set. This latter problem we have just solved by a simple direct randomized algorithm.

References

[1] K. Abrahamson, *On achieving consensus using shared memory*, Proc. 7th ACM Symposium on Principles of Distributed Computing, 1988, pp. 291–302.

[2] Y. Afek, H. Attiya, D. Dolev, E. Gafni, M. Merritt, N. Shavit, *Atomic snapshots of shared memory*, Proc. 9th ACM Symposium on Principles of Distributed Computing, 1990, pp. 1-13.

[3] Y. Afek, E. Gafni, J. Tromp, P.M.B. Vitanyi, Wait-free test-and-set, submitted to WDAG91.

[4] J.H. Anderson, *Composite registers*, Proc. 9th ACM Symposium on Principles of Distributed Computing, 1990, pp. 15-29.

[5] J. Aspnes, *Time- and space-efficient randomized consensus*, Proc. 9th ACM Symposium on Principles of Distributed Computing, 1990, pp. 325-331.

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

[7] D. Dolev and N. Shavit, *Bounded Concurrent Time-Stamp Systems Are Constructible*, Proc. 21th ACM Symposium on Theory of Computing, pp. 454–466, 1989.

[8] M.P. Herlihy, *Impossibility and Universality Results for Wait-Free Synchronization*, Proc. 7th ACM Symposium on Principles of Distributed Computing, 1988, pp. 276–290.

[9] M.P. Herlihy, *Randomized Wait-Free Concurrent Objects*, Proc. 10th ACM Symposium on Principles of Distributed Computing, 1991.

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

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

[12] M. Loui, H.H. Abu-Amara, *Memory requirements for agreement among unreliable asynchronous processes*, pp. 163–183 in: Advances in Computing Research, JAI Press, 1987.

[13] M. Li, J. Tromp, P.M.B. Vitányi, *How to construct concurrent wait-free variables*, Tech. Rept. CS-8916, CWI, Amsterdam, April 1989. See also: pp. 488-505 in: *Proc. International Colloquium on Automata, Languages, and Programming*, Lecture Notes in Computer Science, Vol. 372, Springer Verlag, 1989.

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

[15] G.L. Peterson, M. Fischer, *Economical solutions for the critical section problem in a distributed system*, Proc. 9th ACM Symp. on Theory of Computing, pp. 91-97, 1977.

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

[17] S. Plotkin, *Sticky bits and universality of consensus*, Proc. 8th ACM Symposium on Principles of Distributed Computing, pp. 159-175.

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

[19] M. Saks, N. Shavit, and H. Woll. *Optimal time randomized consensus – making resilient algorithms fast in practice*, Proc. of SODA 90, 1990.

[20] M.O. Rabin, *The choice coordination problem.* Acta Informatica, vol. 17, pp. 121–134, 1982.

[21] 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)