Distrib. Comput. (2002) 15: 127–135

DISTRIBUTED COMPUTING

Randomized two-process wait-free test-and-set

John Tromp, Paul Vitányi

Centrum voor Wiskunde en Informatica, Kuislaan 413, 1098 SJ Amsterdam, The Netherlands (e-mail: {tromp,paulv}@cwi.nl)

Received: June 1991 / Accepted: March 2002

Summary. We present the first explicit, and currently simplest, randomized algorithm for two-process wait-free test-and-set. It is implemented with two 4-valued single writer single reader atomic variables. A test-and-set takes at most 11 expected elementary steps, while a reset takes exactly 1 elementary step. Based on a finite-state analysis, the proofs of correctness and expected length are compressed into one table.

Key words: Test-and-set objects – Symmetry breaking – Asynchronous distributed protocols – Fault-tolerance – Shared memory – Wait-free read/write registers – Atomicity – Randomized algorithms – Adaptive adversary

1 Introduction

A test-and-set protocol concurrently executed by each process out of a subset of n processes selects a *unique* process from among them. In a distributed or concurrent system, the test-and-set operation is useful and sometimes mandatory in a variety of situations including mutual exclusion, resource allocation, leader election and choice coordination. It is well-known that in the wait-free setting, [23], a deterministic construction from atomic read/write variables is impossible [24]. Although widely assumed to exist, and referred to, an explicit randomized construction for wait-free test-and-set has not appeared in print yet, apart from a deterministic construction *assuming* two-process atomic test-and-set [3]. The latter, in the form of a randomized two-process wait-free test-and-set has been circulated in draft form [36] for a decade. Here we finally present the construction. Since such constructions are notoriously prone to hard-to-detect errors, we prove it correct by an exhaustive finite-state proof, thus also presenting a nontrivial application of this proof technique.

Interprocess communication. The model is interprocess communication through shared memory as commonly used in the theory of distributed algorithms [26]. We use atomic single writer single reader registers as primitives. Such primitives can be implemented wait-free from single-reader single-writer "safe" bits (mathematical versions of hardware "flip-flops") [23]). A concurrent object is *constructible* if it can be implemented deterministically with boundedly many safe bits. A deterministic protocol executed by n processes is *wait-free* if there is a finite function f such that every non-faulty process terminates its protocol executing a number of at most $f(n)$ of accesses to the shared memory primitives, regardless of the other processes execution speeds. If the execution speed of a process drops to zero then this is indistinguishable from the process having a crash failure. As a consequence, a wait-free solution can tolerate up to $n-1$ processes having crash failures (a property called "$(n-1)$-resiliency"), since the surviving non-faulty process correctly executes and terminates its protocol. Below, we also write "shared variable" for "register."

Randomization. The algorithms executed by each process are randomized by having the process flip coins (access a random number generator). In our randomized algorithms the answers are always correct—a unique process gets selected— but with small probability the protocol takes a long time to finish. We use the customary assumption that the coin flip and subsequent write to shared memory are separate atomic actions. To express the computational complexity of our algorithm we use the expected complexity, over all system executions and with respect to the randomization by the processes and the worst-case scheduling strategy of an adaptive adversary. A *randomized* protocol is wait-free if $f(n)$ upper bounds the *expectation* of the number of elementary steps, where the expectation is taken over all randomized system executions against the worst-case adversary in the class of adversaries considered (in our results the adaptive adversaries).

Complexity measures. The computational complexity of distributed deterministic algorithms using shared memory is commonly expressed in number and type of intercommunication primitives required and the maximum number of sequential read/writes by any single process in a system execution. Local computation is usually ignored, including coin-flipping in a randomized algorithm.

Partially supported by the EU fifth framework project QAIP, IST–1999–11234, the NoE QUIPROCONE IST–1999–29064, the ESF QiT Programmme, and the EU Fourth Framework BRA NeuroCOLT II Working Group EP 27150.

Related work. 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, for example [30,23,37,22,33,25,32,14,5,2,16]. In contrast, the agreement problem in the deterministic model of computation (shared memory or message passing) is unsolvable in the presence of faults [21,17,24]. Correspondingly, wait-free consensus—viewed as an object on which each of n processes can execute just one operation—is not constructible [12,1], although randomized implementations are possible [12,1,6,34]. Wait-free concurrent test-and-set can deterministically implement two-process wait-free consensus, and therefore is not deterministically constructible [24,17]. This raises the question of whether randomized algorithms for test-and-set exist.

In [17] it is shown that repeated use of 'consensus' on *unbounded* hardware can implement 'test-and-set'. In [31,34,18] it is argued that a bounded solution can be obtained by combining several intermediate constructions, like so-called "sticky bits", but no explicit construction is presented to back up this claim. To quote [31]: "randomized consensus algorithms of Chor, Israeli, and Li [12], Abrahamson [1], Aspnes and Herlihy [7], and Attiya, Dolev, and Shavit [4], together with our construction imply that a polynomial number of safe bits is sufficient to convert a safe implementation into a (randomized) wait-free one." Any such "layered" construction will require orders of magnitude more primitive building blocks like one-writer one-reader bits than the direct construction we present below. Wait-free n-process test-and-set can be implemented deterministically from wait-free two-process test-and-set, [3], showing that the impossibility of a deterministic algorithm for n-process test-and-set is solely due to the two-process case.

Present results. Despite the frequent use of randomized wait-free test-and-set in the literature, no explicit construction for the basic ingredient, randomized wait-free two-process test-and-set, has appeared in print. Our construction, [36], has been subsumed and referred to long since, for example in [3,28,15,10], but other interests prevented us from publishing a final version earlier. The construction is optimal or close to optimal. The presented algorithm directly implements wait-free test-and-set between two processes from single-writer single-reader atomic shared registers. Randomization means that the algorithm contains a branch conditioned on the outcome of a fair coin flip (as in [35]). We use a finite-state based proof technique for verifying correctness and worst-case expected execution length in the spirit of [13]. Our construction is very simple: it uses two 4-valued 1-writer 1-reader atomic variables. The worst-case expected number of elementary steps (called "accesses" in the remainder of the paper) in a test-and-set operation is 11, whereas a reset always takes 1 access.

2 Preliminaries

Processes are sequentially executed finite programs with bounded local variables communicating through single-writer, multi-reader bounded wait-free atomic registers (shared variables). The latter are a common model for interprocess communication through shared memory as discussed briefly in Section 1. For details see [23,25] and for use and motivation in distributed protocols see [8,9,19].

2.1 Shared registers, atomicity

The basic building blocks of our construction are 4-valued 1-writer 1-reader atomic registers. Every read/write register is *owned* by one process. Only the owner of a register can write it, while only one other process can read it. In one *access* a process can either:

- Read the value of a register;
- Write a value to one of its own registers;
- Moreover, following the read/write of a register the process possibly flips a local coin (invokes a random number generator that returns a random bit), preceded or followed by some local computation.

We require the system to be *atomic*: every access of a process can be thought to take place in an indivisible instant of time and in every indivisible time instant at most one access by one process is executed. The atomicity requirement induces in each actual system execution a total order on the set of all of the accesses by the different processes, on the set of accesses of every individual process, and on the set of read/write operations executed on each individual register. The *state* of the system gives for each process: the contents of the program counter, the contents of the local variables, and the contents of the owned shared registers. Since processes execute sequential programs, in each state every process has at most a single access to be executed next. Such accesses are *enabled* in that state.

2.2 Adversary

There is an *adversarial* scheduling demon that in each state decides which enabled access is executed next, and thus determines the sequence of accesses of the system execution. There are two main types of adversaries: the *oblivious* adversary that uses a fixed schedule independent of the system execution, and the much stronger *adaptive* adversary that dynamically adapts the schedule based on the past initial segment of the system execution. Our results hold against the adaptive adversary—the strongest adversary possible.

2.3 Complexity

The computational complexity of a randomized distributed algorithm in an adversarial setting and the corresponding notion of wait-freeness require careful definitions. For the rigorous novel formulation of adversaries as restricted measures over the set of system executions we refer to the Appendix of [28]. For the simple application in this paper we can assume that the notions of global (system) execution, wait-freeness, adaptive adversary, and expected complexity are familiar. A randomized distributed algorithm is *wait-free* if the expected number of read/writes to shared memory by every participating process is bounded by a finite function $f(n)$, where n is the number of processes. The expectation is taken over the probability measure over all randomized global (system) executions against the worst-case adaptive adversary.

3 Test-and-set implementation

We first specify the semantics of the target object:

Definition 3.1. *An* atomic test-and-set object X *is a* global variable*, associated with n processes $P_0, \dots, P_{n-1}$, exhibiting the following functionality:*

- *The value of X is 0 or 1;*
- *Every process P_i has a* local binary variable x_i *which it alone can read or write;*
- *At any time exactly one of $X, x_0, \dots, x_{n-1}$ has value 0, all others have value 1 (we assume the global time model);*
- *A process P_i with $x_i = 1$ can atomically execute a* test-and-set *operation τ:*
  read $x_i := X$; write $X := 1$; return x_i.
- *A process P_i with $x_i = 0$ can atomically execute a* reset *operation ρ:*
  $x_i := 1$; write $X := 0$.

This specification naturally leads to the definition of the state *of the test-and-set object as an element of $\{\bot, 0, \dots, n-1\}$ corresponding to the unique local variable out of $X, x_0, \dots, x_{n-1}$ that has value 0. Here $\bot$ is the state that none of the x_i's is 0. Formally, the specification is given later as a finite automaton in Definition 5.1.*

Since "atomicity" means that the operation is executed in a single indivisible time instant, and, moreover, in every such time instant at most one operation execution takes place, the effect of a test-and-set operation by process P_i is that $x_i := 0$ iff $x_j \neq 0$ for all $j \neq i$, and $x_i = 1$ otherwise. The effect of a reset operation by P_i is only defined for initially $x_i = 0$ and $x_j \neq 0$ for all $j \neq i$, and results in $x_i := 1$. To synthesise the target object from more elementary objects, we have to use a sequence of atomic accesses to these elementary objects. By adversary scheduling, these sequences may be interleaved arbitrarily. Yet we would like to have the effect of an atomic execution of the test-and-set operations and the reset operations by each process. To achieve such a "virtual" atomic execution we proceed as follows:

Definition 3.2. *An* implementation *of a* test-and-set *operation τ or a* reset *operation ρ by a process P is an algorithm executed by P that results in an ordered sequence of accesses of that process to elements of a set $\{R_0, \dots, R_{m-1}\}$ of atomic shared variables, interspersed with local computation and/or local coin flips. The sequence of accesses is determined by the, possibly randomized, algorithm, and the values returned by the "read" accesses to shared variables. We denote an access by (P, R, A), meaning that process P executes access A (read or write a value) on shared variable R. The implementation must satisfy the specification of the target test-and-set semantics of Definition 3.1 restricted to process P. Formally, the specification is given later as a finite automaton in Definition 5.2.*

Definition 3.3. *A* local execution *of a process P consists of the (possibly infinite) sequence of test-and-set operations and reset operations it executes, according to the implementation, each such operation $a \in \{\tau, \rho\}$ provided with a* start *time $s(a)$ and a* finish *time $f(a)$—we assume a global time model. Note that $s(a)$ coincides with the time of execution of the first access in the ordered sequence constituting a, and $f(a)$ coincides with the time of execution of the last access in the ordered sequence constituting a. By the atomicity of the individual accesses in the global time model, all accesses are executed at different time instants. In certain cases (which we show to have zero probability) it is possible that $f(a)$ is not finite (because the algorithm executes infinitely many loops with probability $\frac{1}{2}$ each).*

Definition 3.4. *Let the local execution of process P_i consist of the ordered sequence of operations $a_1^i, a_2^i, \dots$ $(0 \leq i \leq n-1)$. A* global execution *consists of the pair $(\mathcal{A}, \rightarrow)$ where $\mathcal{A} = \{a_j^i : j = 1, 2, \dots, \ 0 \leq i \leq n-1\}$ and $\rightarrow$ is a partial order on the elements of $\mathcal{A}$ defined by $a \rightarrow b$ iff $f(a) < s(b)$ (the last access of a precedes the first of b). We require that the number of b such that $b \rightarrow a$ is finite for each a.*

A test-and-set operation or reset operation by a particular process may consist of more than one access, and therefore the local executions by the different processes may happen concurrently and asynchronously. This has the effect that a *global execution* can correspond to many different interleavings.

Definition 3.5. *Consider a global execution. An* interleaving *of the accesses by the different processes associated with the global execution is a (possibly infinite) totally ordered sequence $(P^1, R^1, A^1), (P^2, R^2, A^2) \dots$, where (P^i, R^i, A^i) is the ith access, respecting*

- *The start times and finish times determined by the local executions; and*
- *the order of the accesses in the local executions.*

The implementation should guarantee that the functionality of the implementation is "equivalent", in an appropriate sense, to the functionality of the target test-and-set object, and in particular satisfies the "linearizability requirement" [20] (also called "atomicity" in [23]).

Definition 3.6. *A system* implements *the target test-and-set object if the system is initially in state $\bot$, and we can extend $\rightarrow$ on $\mathcal{A}$ to a total order $\Rightarrow$ on $\mathcal{A}$ with an initial element, satisfying:*

- *From state $\bot$, a successful test-and-set operation τ executed by process P_i (setting $x_i := 0$) moves the system to state i at some time instant in the interval $[s(\tau), f(\tau)]$;*
- *from state i, a reset operation ρ executed by process P_i moves the system to state $\bot$ at some time instant in the interval $[s(\rho), f(\rho)]$;*
- *From state i, every operation execution different from a reset by process P_i leaves the system invariant in state i; and*
- *No other state transitions than the above are allowed.*

The implementation must satisfy the specification of the target test-and-set semantics of Definition 3.1. Formally, the specification is given later as a finite automaton in Definitions 5.3 and 5.4.

To prove that a protocol executed by all processes is an implementation of the target test-and-set object it suffices to show that every possible interleaving that can be produced by the processes executing the protocol in every global execution, starting from the $\bot$ state, satisfies the above requirements.

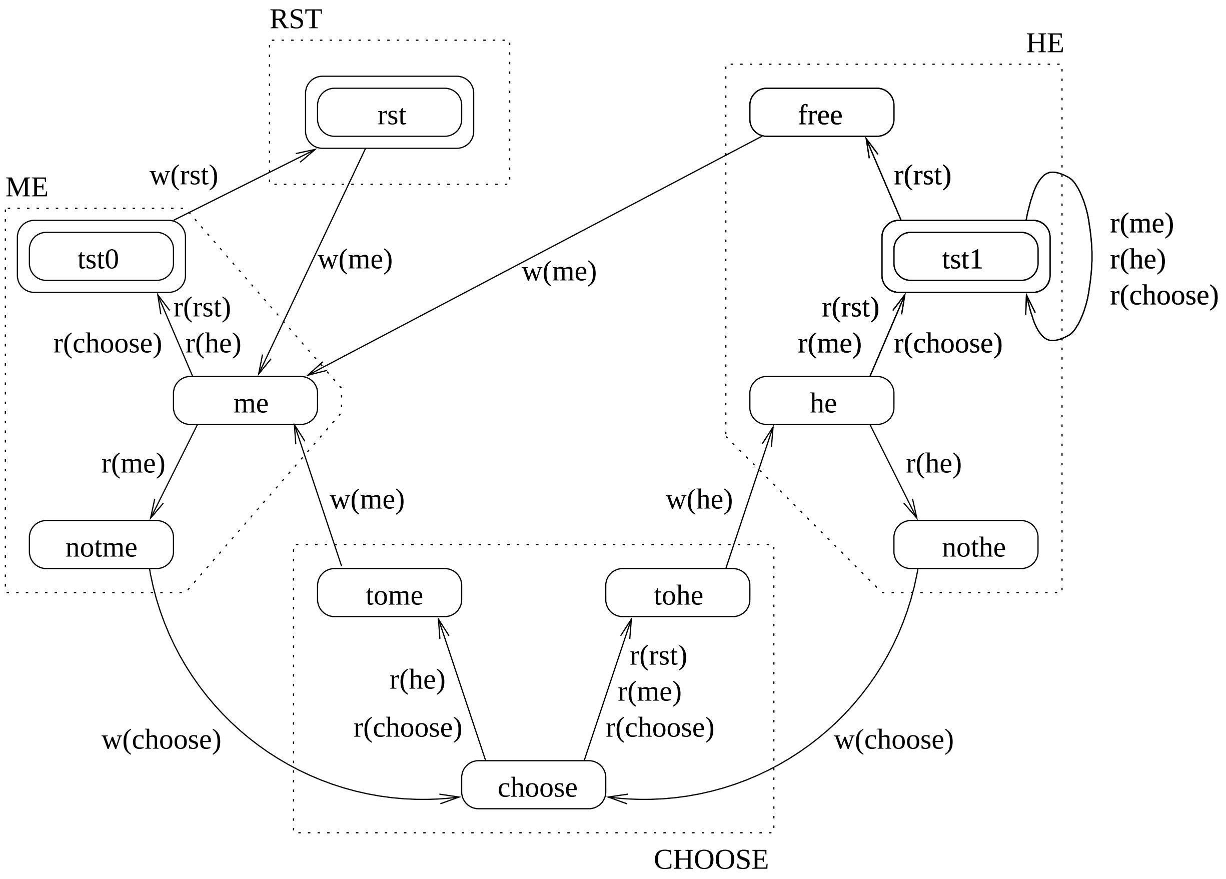

Fig. 1. State Chart

4 Algorithm

We give a test-and-set implementation between two processes, process P_0 and process P_1. The construction uses two 4-valued shared read/write variables R_0 and R_1. The four values are 'me', 'he', 'choose', 'rst'—chosen as a mnemonic aid explained below. Process P_i solely writes 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 P_i $(i = 0, 1)$, is presented as both a finite state chart, Fig. 1 and as the program below. The state chart representation will simplify the analysis later. The transitions in the state chart 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 state chart are split into 4 groups enclosed by dotted lines. Each group is an equivalence class consisting of the set of states in which process P_i's own shared variable R_i has the same value. That is, the states in a group are equivalent in the sense that process P_{1-i} cannot distinguish between them by reading R_i. Accordingly, the inter-group transitions are writes to R_i, whereas the intra-group transitions are reads of R_{1-i}. Each group is named after the corresponding value of the own shared variable R_i. The state chart is deterministic, but for a coin flip which is modeled by the two inter-group transitions in the "choose" group, representing the two outcomes of a fair coin flip. Doubly circled states are "idle" states (no operation execution is in progress), and singly circled states are intermediate states in an operation execution that is in progress.

A program representation of the protocol, for process P_i, is given below. An occurrence of R_i not preceded by 'write' (similarly, R_{1-i} not preceded by 'read') as usual refers to the last value written to it (resp. read from it). The conditional 'rnd(true,false)' represents the boolean outcome 'true' or 'false' of a fair coin flip. The system is initialized with value 'rst' in shared variables R_0, R_1. In our protocol, all assignments to local variables consist of contents read from shared variables. To simplify, we abbreviate statements like "while $((r_{1-i} := R_{1-i}) = r_i)$ do"; to "while read $R_{1-i} = R_i$ do … ". Here, r_i is the local variable containing the value last written to shared variable R_i and r_{1-i} is the local variable storing the last read value of shared variable R_{1-i}, for process P_i. This way, our (writing of the) protocol can dispense with local variables altogether.

```
test_and_set:

if $R_i =$ he AND read $R_{1-i} \neq$ 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 state chart represents the operation of the program. The intuition is easily explained using the state chart. The default situation is where both processes are idle, which corresponds to being in the ‘rst’ state. If process P_i starts a test-and-set then it writes $R_i :=$ me (indicating its desire to take the 0), and checks by reading R_{1-i} whether process P_{1-i} agrees (by *not* having $R_{1-i} =$ me). If so, then P_i has successfully completed a test-and-set by obtaining the 0 and, implicitly, setting the global variable $X := 1$. In this case process P_{1-i} cannot get 0 until process P_i does a reset by writing $R_i :=$ rst. While $R_i =$ me, process P_{1-i} can only move from state ‘me’ to state ‘notme’ and on via states ‘choose’, ‘tohe’ and ‘he’ to ‘tst1’, where it completes its test-and-set operation by failure to obtain the 0.

The only complication arises if both processes see each other’s variable equal to ‘me’. In this case they are said to *disagree* or to be *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, 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 is read to contain a value 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 resulting in obtaining the 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 it resumes trying to take the 0 itself.

Suppose now that process P_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 P_i will move to the opposite state as did process P_{1-i}, and thus the conflict will be resolved.

In the proof of Theorem 5.13 (below) we establish that the probability of each loop through the ‘choose’ state is at most one half, and the expected number of ‘choices’ (transitions from state choose) is at most two. This indicates that the worst case expected test-and-set length is 11. Namely, starting from the ‘tst1’ state, it takes 4 accesses to get to state ‘choose’, another 4 accesses to loop back to ‘choose’ and 3 more accesses to reach ‘tst0’/‘tst1’. The reset operation always takes 1 access.

5 Proof of Correctness

The proof idea is as follows: We give a specification of a correct implementation of two-process test-and-set in the form of a finite automaton (Fig. 4). We then show that all initial segments of every possible interleaving of accesses by two processes P_0 and P_1, both executing the algorithm of the state chart (Fig. 1), are accepted by the finite automaton. Moreover, the sequence of states of the finite automaton in the acceptance process induces a linear order on the operation execution of the implemented processes that extends the partial order induced by the start and finish times of the individual operation executions. Thus, the implementation is both correct and atomic. Essentially, the proof is given by Fig. 5, which gives the state of the specification finite automaton for every reachable combination of states which processes P_0 and P_1 can attain in their respective copies of the state chart (Fig. 1). By analysis of the state chart, or Fig. 5, we upper bound the expectation of the number of accesses of every operation execution of the implementation by a small constant. Hence the implementation is wait-free.

Let h be an interleaving corresponding to a global execution $(\mathcal{A}, \rightarrow)$ of two processes running the protocol starting from the initial state. Let $\{s(a), f(a) : a \in \mathcal{A}\}$ be the set of time instants that start or finish an operation execution, each such time instant corresponding to an access (P, R, A). Let B denote the set of these accesses. Recall that if a is a reset, then we have $s(a) = f(a)$ and there is but a single access executing this operation.

By definition, $h|B$, the restriction of h to the accesses in B, completely determines the partial order $\rightarrow$. If, for every $a \in A$ we can choose a single access $(P, R, A)_a$ in the sequence of accesses constituting the operation execution of a, such that if $a \rightarrow b$ then $(P, R, A)_a$ precedes $(P, R, A)_b$ in h, then we are done. Namely, we can imagine an operation a as executing atomically at the time instant of atomic access $(P, R, A)_a$, and the total order $\Rightarrow$ defined by $a \Rightarrow b$ iff $(P, R, A)_a$ precedes $(P, R, A)_b$ in h, extends the partial order $\rightarrow$. Denote the set $\{(P, R, A)_a : a \in \mathcal{A}\}$ by C. We have to show that for every h as defined above such a C can be found.

Definition 5.1. Specification of two-process atomic test-and-set: *The definition of the target atomic test-and-set for two processes, process P_0 and process P_1, is captured by finite automaton FA1 in Fig. 2, which accepts all possible sequences of atomic test-and-set and reset operations (all states final). The states are labeled with the owner of the 0-bit. The arcs representing actions of process P_1 are labeled, whereas the non-labeled arcs represent the corresponding actions of process P_0: resulting in setting $x_1 := 1$.*

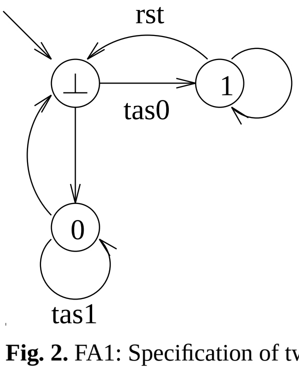

Fig. 2. FA1: Specification of two-process atomic test-and-set object

Definition 5.2. Specification of wait-free atomic test-and-set restricted to a single process: *Fig. 3 shows the semantics required of a correct implementation of a wait-free test-and-set* *object as a finite automaton FA2, that accepts all sequences of accesses by a single process P_i ($i = 0, 1$) executing a correct wait-free atomic test-and-set protocol: (all states final):*

- *the access starting a test-and-set operation execution, denoted s(tas),*
- *the atomic occurrence of a test-and-set operation execution returning 0, denoted tas0,*
- *the atomic occurrence of a test-and-set operation execution returning 1, denoted tas1,*
- *the access finishing a test-and-set operation execution returning 0, denoted f(tas0),*
- *the access finishing a test-and-set operation execution returning 1, denoted f(tas1),*
- *the single access corresponding to a complete reset operation execution, denoted rst.*

These are the events in $B \cup C$ restricted to a process P_i. The reason for not splitting a reset operation execution into start, atomic occurrence, and finish is that it is implemented in our protocol as a single atomic write where the above three transitions coincide. As before, doubly circled states are "idle" states (no operation execution is in progress), and singly circled states are intermediate states in an operation execution that is in progress.

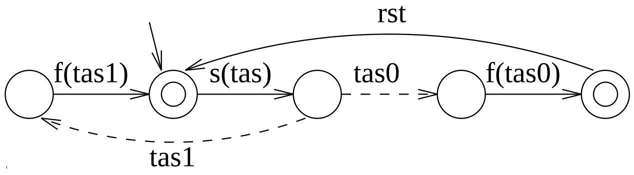

Fig. 3. FA2: Specification of 1-process wait-free implementation of atomic test-and-set

Definition 5.3. Specification of two-process wait-free atomic test-and-set: *The proof that our implementation is correct consists in demonstrating that it satisfies the specification in the form of the finite automaton FA3 in Fig. 4 below (again all states are final).*

Formally [27], FA3 is the composition of FA1 with two copies of FA2, in the I/O Automata framework, as follows: It is drawn as a cartesian product of the two component processes – transitions of process P_0 are drawn vertically and those of process P_1 horizontally. For clarity, the transition names are only given once: only for process P_1. Identifying the starts and finishes of test-and-set operation executions a with their atomic occurrence $(P, R, A)_a$ by collapsing the s() and f() arcs, FA3 reduces to the atomic test-and-set diagram FA1. Identifying all nodes in the same column (row) reduces FA3 to FA2 of process P_0 (process P_1).

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

The broken transitions of Fig. 4 correspond to the access $(P, R, A)_a \in C$, required for a correct implementation, where the atomic execution of operation a can be virtually situated. Recall that this is only relevant for a is a test-and-set operation, since the reset operation is implemented in the protocol already in a single atomic access of a shared primitive variable.

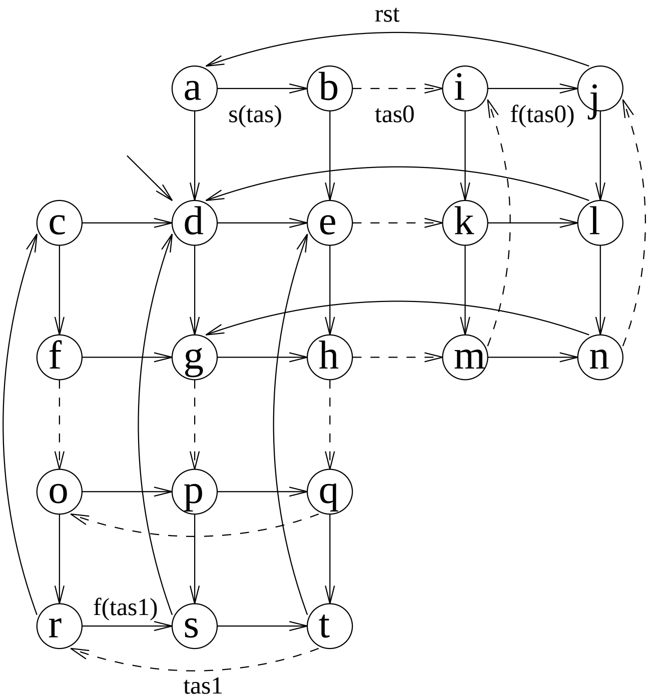

Fig. 4. FA3/FA4: Specification of two-process wait-free atomic test-and-set

Definition 5.4. *Let FA4 be the (nondeterministic) finite automaton obtained from FA3 by turning the broken transitions of Fig. 4, which correspond to the unknown but existing access $(P, R, A)_a \in C$ where the execution of a can be virtually situated, into ϵ-moves.*

Lemma 5.5. *Acceptance of $h|B$ by FA4 implies that $(\mathcal{A}, \rightarrow)$ is linearizable: the partial order $\rightarrow$ can be extended to a total order $\Rightarrow$ such that the sequence of operation executions in $\mathcal{A}$ ordered by $\Rightarrow$ satisfy the test-and-set semantics specification of Definition 5.1.*

Proof. If FA4 accepts $h|B$, then, corresponding to the ϵ moves, we can augment the sequence $h|B$ with an access $(P, R, A)_a$ in the interval $[s(a), f(a)]$ of each operation execution $a \in \mathcal{A}$ – or select the single access involved if $s(a) = f(a)$ as in the case of a reset operation execution – to obtain a new sequence h' that is accepted by FA3. By the way FA1 composes FA3, it accepts $h'|C$, the subsequence of atomic accesses $(P, R, A)_a$ with $a \in \mathcal{A}$ contained in h'. Furthermore, letting $t(a)$ denote the time of access $(P, R, A)_a$, we have $a \rightarrow b$ iff $t(a) \leq f(a) \leq s(b) \leq t(b)$. Defining $a \Rightarrow b$ if $t(a) < t(b)$, the total order of accesses in $h'|C$, then $\Rightarrow$ is a total order that extends the partial order $\rightarrow$. That is, the sequence of operation executions of $\mathcal{A}$, linear ordered by $\Rightarrow$, is accepted by FA1.

Recall that Fig. 1 is the state chart of the execution of the implementation of an operation by a single process. Each process can be in a particular state of the state chart. Let (s_0, s_1) denote the state of the system with process P_i in state s_i ($i \in \{0, 1\}$).

Definition 5.6. *The* initial *system state is* (rst, rst)*. A system state* (s_0, s_1) *is* reachable *from the initial system state*

	rst	tst0	notme	me	tome	choose	tohe	he	nothe	tst1	free
rst	d**10**	l**10**	cek**10**	ek**10**	ek**10**	c**10**	c**10**	c**10**	c**10**	d**10**	ek**10**
tst0	s**1**	*	rt**1**	rt**1**	rt**1**	r**1**	r**1**	r**1**	r**1**	s**1**	rt**1**
notme	agp**8**	jn**8**	imoq**8**	imoq**8**	*	imoq**8**	imoq**8**	o**4**	*	p**4**	*
me	gp**9**	jn**9**	imoq**9**	imoq**9**	imoq**9**	o**1**	o**1**	o**1**	o**1**	p**1**	imoq**9**
tome	gp**10**	jn**10**	*	imoq**10**	imoq**10**	imoq**6**	o**2**	o**2**	imoq**6**	p**2**	*
choose	a**3**	j**3**	imoq**7**	i**3**	imoq**7**	imoq**7**	imoq**7**	o**3**	imoq**7**	p**3**	*
tohe	a**2**	j**2**	imoq**6**	i**2**	i**2**	imoq**6**	imoq**10**	imoq**10**	*	p**6**	*
he	a**1**	j**1**	i**1**	i**1**	i**1**	i**1**	imoq**9**	imoq**9**	imoq**9**	p**5**	*
nothe	a**4**	j**4**	*	i**4**	imoq**8**	imoq**8**	*	imoq**8**	imoq**8**	p**4**	*
tst1	d**11**	l**11**	k**11**	k**11**	k**11**	k**11**	k**11**	k**11**	k**11**	*	*
free	gp**10**	jn**10**	*	imoq**10**	*	*	*	*	*	*	*

Fig. 5. Table verification of correctness and wait-freedom

(rst, rst) *if there is a sequence* h *arising from the execution of our test-and-set implementation, represented by the state chart of Fig. 1, starting from the initial state and ending in state* (s_0, s_1).

Example 5.7. In the initial state both processes are in state 'rst'. Process P_0 can start a test-and-set by executing $w(me)$ and entering state me. Suppose process P_1 now starts a test-and-set: it executes $w(me)$ and moves to state me. Hence, system states (me, rst) and (me, me) are reachable states.

Definition 5.8. *The* representative set *of a reachable system state* (s_0, s_1) *is a nonempty set* S_{s_0,s_1} *of FA3/FA4 states, as in Fig. 4, such that: For every sequence of accesses* h *starting in the initial state and ending in state* (s_0, s_1)*, the set* S_{s_0,s_1} *is the set of states in which FA4 can be after processing* $h|B$*, excluding those states that have outgoing moves that are* ϵ*-moves only.*

Example 5.9. We elaborate Example 5.7. In the initial state both processes are in state 'rst'. The corresponding start state d of FA4 gives the associated (in this case singleton) representative set $\{d\}$. When process P_0 executes $w(me)$ and enters state me, the resulting system state is (me, rst) with the associated representative set $\{g, p\}$ of FA4 states. That is, the system is now either in state g, meaning that process P_0 has executed $s(tas)$, or in state p meaning that process P_0 has executed $s(tas)$ and also $tas0$ atomically. In the scenario of Example 5.7, process P_1 now executes $w(me)$ and moves to state me, resulting in the system state (me, me). The corresponding representative set of FA4 states is $\{i, m, o, q\}$. State m says process P_1 has executed $s(tas)$ and $tas0$ atomically, while process P_0 has only executed $s(tas)$ – hence the system was previously in state g and not in state p. State i says process P_1 has executed $s(tas)$ and $tas0$ atomically, while process P_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 P_0 and process P_1 interchanged, and the previous system state is either p or g. (The correspondence between reachable states and their representative sets is exhaustively established in Claim 5.11 below.)

Lemma 5.10. *Let* h *be a sequence of accesses arising from the execution of our test-and-set implementation, represented by the state chart of Fig. 1, starting from the initial state (both processes in state 'rst'). Then, every initial segment of* $h|B$ *is accepted by FA4 starting from initial state 'd'.*

Proof. We show that the set of letters in an entry in the table of Fig. 5 is a representative set for the state of process P_0, indexing the row, and the state of process P_1, indexing the column. The entries were chosen excluding all states from the representative sets with all outgoing moves consisting of ϵ-moves (but the representative sets contain the states the outgoing ϵ-moves of the excluded states point to). This gives the most insight into the workings of the protocol by considering only the result of executing ϵ-moves from a state if its only outgoing moves are ϵ-moves. A $*$-entry indicates an unreachable state pair. (The number ending an entry gives the expected number of accesses to finish the current operation execution of process P_0 – and by symmetry, that for an equivalent state pair with respect to P_1. We will use this later.) Thus, every state (s_0, s_1) of the implementation execution corresponds with a set of states S_{s_0,s_1} of FA4.

Claim 5.11. The representative sets are given by the entries of Fig. 5.

Proof. The proof of the claim is contained in the combination of Figs. 1, 4, 5. Below we give the inductive argument. The mechanical verification of the subcases has been done by hand, and again by machine. The setting up of the exhaustive list subcases and subsequent verification by a computer program is the essennce of a finite-state proof. In this particular case, exceptionally, the finite state machines involved (and the table of representative sets) have been minimized so that "mechanical" verification by hand by the reader is still feasible. Induction is on the length of the sequence of accesses:
Base Case. Initially, after an empty sequence of accesses, FA4 is in the state $\{d\} = S_{rst,rst}$.
Induction. Every non-reachable state has a $*$-entry in the table of Fig.5. Consider an arbitrary atomic transition from a reachable state (s_0, s_1) to a state (t_0, t_1), that is, using a single arc in the state chart in Fig. 1 for either process P_0 or P_1. This way, either $t_0 = s_0$ or $t_1 = s_1$ but not both. Then, for every FA4 state $y \in S_{t_0,t_1}$, Fig. 4, according to the table of Fig. 5, there is an FA4 state $x \in S_{s_0,s_1}$ according to Fig. 4, such that FA4 can move from x to y by executing: either the access corresponding to the transition in the state chart in Fig. 1, if that access belongs to B, or no access otherwise (there is a sequence of ϵ-moves from x to y). This finalizes the proof of the claim.

Since every reachable state of the system (s_0, s_1), with s_i ($i \in \{0, 1\}$) a state of the state chart of Fig. 1, has a representative set in FA4, Fig. 4, and every state of of FA4 is an accepting state, the lemma follows from Claim 5.11.

Theorem 5.12. *The algorithm represented by state chart of Fig. 1 correctly implements an atomic test-and-set object.*

Proof. By Lemma 5.10 the implementation by the state chart in Fig. 1 correctly implements the specification of two-process test-and-set given by Fig. 4. The implementation is linearizable (atomic) by Lemma 5.10. The system makes progress (every operation execution is executed completely except for possibly the last one of each process) since $h|B$ contains only the start and finish accesses of each operation execution performed by the implementation.

Theorem 5.13. *The algorithm represented by state chart of Fig. 1 is wait-free: the expected number of accesses to shared variables never exceeds 11 during execution of an operation.*

Proof. In Fig. 1 every arc is an access. Double circled states are idle states (in between completing an operation execution and starting a new one). Consider process P_0 (the case for process P_1 is symmetrical). The longest path without completing an operation and without cycling is from state 'tst1': tst1, free, me, notme, choose, tohe, he, tst1. This takes 7 accesses. Four of these accesses are parts of a potential cycle of length 4. The remainder is 3 accesses outside the potential cycle. In state 'choose', the outgoing arrow is a random choice only when process P_1 is also in the CHOOSE group. If it is, then with $\frac{1}{2}$ probability P_1 makes (or has already made) a choice which will cause process P_0 to loop back to the 'choose' state again. This can happen again and again. The expected number of iterations of loops is $\sum_{i=1}^{\infty} i\left(\frac{1}{2}\right)^i = \frac{1}{2}(1-\frac{1}{2})^{-2} = 2$. Since a loop has length 4, this gives a total of expected accesses of 8 for the loops. Together with 3 non-loop accesses the total is at most 11 accesses. Such a computation holds for every state in the state chart of Fig. 1, the only loop being the one discussed but the longest possible path is the one starting from 'tst1'. For definiteness, we have in fact computed the expected number of accesses for every accessible state (s_0, s_1) according to the state chart of Fig. 1, and added that number to the representative set concerned in the table of Fig. 5. Since the expected number of accesses is between 1 and 11 for all operation executions, the algorithm given by the state chart of Fig. 1 is wait-free.

To aid intuition, we give an example of checking a few transitions below, as well as giving the interpretation.

Example 5.14. We elaborate and continue Examples 5.7, 5.9. In the initial state both processes are in state 'rst'. In Fig. 4, the table entry $d10$ gives the corresponding start state d of FA4. The worst-case expected number of accesses for a test-and-set by process 0 is 10. Process P_0 can start a test-and-set by executing $w(me)$ and entering state me. The corresponding table entry $gp9$ indicates in Fig. 4 that the system is now either in state g meaning that process P_0 has executed $s(tas)$, or in state p meaning that process P_0 has executed $s(tas)$ and also $tas0$ atomically. The expected number of accesses is now $9 \leq 10 - 1$. Suppose process P_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 Fig. 4 and the expected number of accesses for execution of test-and-set by process P_0 is still 9. State m says process P_1 has executed $s(tas)$ and $tas0$ atomically, while process P_0 has only executed $s(tas)$ – hence the system was previously in state g and not in state p. State i says process P_1 has executed $s(tas)$ and $tas0$ atomically, while process P_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 P_0 and process P_1 interchanged, and the previous system state is either p or g.

Note that at this point the system can also be in state h of FA4 – both processes having executed $s(tas)$ but no process having executed $tas0$ or $tas1$. However, from h there are two ϵ-moves possible, and no other moves, leading to q and m. This corresponds to the fact that if both processes have executed $s(tas)$, one of them must return 0 and the other one must return 1. We have optimized the table entries by eliminating such spurious intermediate states h with outgoing moves that are ϵ-moves only.

Process P_0 might now read $R_1 = me$, and move via state 'notme' (table entry $imoq8$) by writing $R_0 := choose$, to state 'choose'. Process P_1 is idle in the meantime. The table entry is now $i3$. This says that process P_1 has atomically executed $tst0$, and process P_0 has atomically executed $tst1$. Namely, all subsequent schedules lead in 3 accesses of process P_0 to state 'tst1' – hence the expectation 3.

The expected number of remaining accesses of process P_0's test-and-set has dropped from 8 to 3 by the last access 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 P_1 to move to $(notme, notme)$ with table entry $imoq8$, followed by a move of process P_1 to state $(notme, choose)$ with table entry $imoq8$, followed by a move of process P_0 to state $(choose, choose)$ with table entry $imoq7$. Suppose the adversary now schedules process P_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 access of process P_0 correctly decrements the expected number of accesses. Suppose the adversary schedules process P_1 in state $(choose, choose)$. Process P_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 $imoq7$.

6 Remark on multi-process test-and-set

The obvious way to extend the given solution to more than two processes would be to arrange them at the leaves of a binary tree. Then, a process wishing to execute an n-process test-and-set, would enter a tournament, as in [29], by executing a separate two-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 $\log n$ (binary logarithm) times more than that of the two-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 two-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 two-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 Symp. Principles of Distributed Computing, 291–302, 1988
2. Y. Afek, H. Attiya, D. Dolev, E. Gafni, M. Merritt, N. Shavit, Atomic snapshots of shared memory, Journal of the ACM 40(4): 873–890 (1993)
3. Y. Afek, E. Gafni, J. Tromp, P.M.B. Vitányi, Wait-free test-and-set, pp. 85-94 in: Proc. 6th Workshop on Distributed Algorithms (WDAG-6), Lecture Notes in Computer Science, vol. 647, Berlin: Springer 1992
4. H. Attiya, D. Dolev, N. Shavit, Bounded Polynomial Randomized Consensus, Proc.8th ACM Symp. Principles of Distributed Computing, 281–293, 1989
5. J.H. Anderson, Multiwriter composite registers, Distributed Computing 7(4): 175–195 (1994)
6. J. Aspnes, Time- and space-efficient randomized consensus, Journal of Algorithms 14(3): 414–431 (1993)
7. J. Aspnes, M. Herlihy, Fast randomized consensus using shared memory, Journal of Algorithms 11(3): 441–461 (1990)
8. A. Bar-Noy, D. Dolev, A partial equivalence between shared-memory and message-passing in an asynchronous fail-stop distributed environment, Mathematical Systems Theory 26: 21–39 (1993)
9. E. Borowsky, E. Gafni, Immediate Atomic Snapshots and Fast Renaming, Proc. 11th ACM Symp. on Principles of Distributed Computing, pp. 41–52, 1992
10. H. Buhrman, A. Panconesi, R. Silvestri, P. Vitányi, On the importance of having an identity or, is consensus really Universal? Distributed Computing Conference (DISC'00), Lecture Notes in Computer Science, Vol. 1914, 134–148. Berlin: Springer 2000
11. J.E Burns, P.Jackson, N.A. Lynch, M.J. Fischer, G.L. Peterson, Data Requirements for Implementation of N-process Mutual Exclusion Using a Single Shared Variable, J. Assoc. Comput. Mach. 29: 183–205 (1982)
12. B. Chor, A. Israeli, M. Li, Wait–Free Consensus Using Asynchronous Hardware, SIAM J. Comput. 23(4): 701–712 (1994)
13. E. M. Clarke, O. Grumberg, D. Peled, Model Checking, MIT Press, 2000
14. D. Dolev, N. Shavit, Bounded concurrent time-stamp systems are constructible, Siam J. Comput. 26(2): 418-455 (1997)
15. W. Eberly, L. Higham, J. Warpechowska-Gruca, Long-lived, fast, waitfree renaming with optimal name space and high throughput, Proc. 12th Intn'l Distributed Computing Conference (DISC'98), Lecture Notes in Computer Science, 1499, Berlin: Springer 1998
16. S. Haldar, P.M.B. Vitányi, Bounded concurrent timestamp systems using vector clocks, J. Assoc. Comput. Mach. 49(1): (2002)
17. M. Herlihy, Wait-free synchronization. ACM Trans. Progr. Lang. Syst. 13(1): 124–149 (1991)
18. M.P. Herlihy, Randomized Wait-Free Concurrent Objects, Proc. 10th ACM Symp. Principles of Distributed Computing 11–21, 1991
19. M. Herlihy, N. Shavit, The topological structure of asynchronous computability, J. Assoc. Comp. Mach. 46(6): 858–923 (1999)
20. M. Herlihy, J. Wing, Linearizability: A correctness condition for concurrent objects, ACM Trans. Program. Languages and Systems, 12: 463–492 (1990)
21. M.J. Fischer, N.A. Lynch, and M.S. Paterson, Impossibility of Distributed Consensus with One Faulty Processor. J. Assoc. Comput. Mach. 32(2): 374–382 (1985)
22. A. Israeli, M. Li, Bounded Time-Stamps, Distributed Computing 6: 205–209 (1993)
23. L. Lamport, On Interprocess Communication Parts I and II, Distributed Computing 1: 77–101 (1986)
24. M. Loui, H.H. Abu-Amara, Memory requirements for agreement among unreliable asynchronous processes, pp. 163–183. In: Advances in Computing Research, Vol. 4, JAI Press, 1987
25. M. Li, J. Tromp, P.M.B. Vitányi, How to share concurrent wait-free variables, J. Assoc. Comp. Mach. 43: 723–746 (1996)
26. N.A. Lynch, Distributed Algorithms, Morgan Kaufmann, 1996
27. N.A. Lynch, M. Tuttle, An Introduction to Input/Output automata, CWI-Quarterly 2(3): 219–246 (1989)
28. A. Panconesi, M. Papatrintafilou, P. Tsigas, P. Vitányi, Randomized Naming Using Wait-Free Shared Variables, Distributed Computing 11: 113–124 (1998)
29. G.L. Peterson, M. Fischer, Economical solutions for the critical section problem in a distributed system, Proc. 9th ACM Symp. Theory of Computing, 91–97, 1977
30. G.L. Peterson, Concurrent reading while writing, ACM Trans. Programming Languages and Systems, 5(1): 46–55 (1983)
31. S. Plotkin, Sticky bits and universality of consensus, Proc. 8th ACM Symp. Principles of Distributed Computing, 159–175, 1989
32. A.K. Singh, J.H. Anderson, M.G. Gouda, The Elusive Atomic Register Revisited, J. Assoc. Comput. Mach. 41(2) 311–339 (1994)
33. R. Schaffer, On the correctness of atomic multi-writer registers, Technical Report MIT/LCS/TM-364, MIT lab. for Computer Science, June 1988
34. M. Saks, N. Shavit, H. Woll, Optimal time randomized consensus – making resilient algorithms fast in practice, 2nd ACM Symp. On Discrete Algorithms, 351–362, 1991
35. M.O. Rabin, The choice coordination problem. Acta Informatica 17: 121–134 (1982)
36. J. Tromp, P. M. B. Vitányi, Randomized wait-free test-and-set, Manuscript, November 1990
37. P.M.B. Vitányi, B. Awerbuch, Atomic Shared Register Access by Asynchronous Hardware, Proc. 27th IEEE Symp. Foundations of Computer Science, 233–243, 1986 (Errata, Ibid.,1987)