Fast Timing-based Algorithms*

Rajeev Alur † Gadi Taubenfeld‡

Abstract

Concurrent systems in which there is a known upper bound Δ on memory access time are considered. Two prototypical synchronization problems, mutual exclusion and consensus, are studied and solutions that have constant (i.e. independent of Δ and the total number of processes) time complexity in the absence of contention are presented. For mutual exclusion, in the absence of contention, a process needs only five accesses to the shared memory to enter its critical section, and in the presence of contention, the winning process may need to delay itself for $4 \cdot \Delta$ time units. For consensus, in absence of contention, a process decides after four accesses to the shared memory, and in the presence of contention, it may need to delay itself for Δ time units.

1 Introduction

The possibility and complexity of synchronization in a distributed environment depends heavily on timing assumptions. In the asynchronous model no timing assumptions are made about the relative speeds of the processes, while a timing-based model assumes known bounds on the speeds of the processes. In this paper, we study two prototypical coordination and synchronization problems, namely, mutual exclusion and consensus, in the latter model.

We use a shared memory model where processes communicate with each other by reading and writing to shared registers. We assume that there is an upper bound, denoted by Δ, on time required for a single access to shared memory. There is no lower bound on time needed to execute a step, but a process can delay itself explicitly by executing a statement *delay*(d), for some constant d. Furthermore, we assume that the value of Δ is known to all the processes, and thus, a process, by executing a statement *delay*(Δ), can ensure that every process currently accessing shared memory takes at least one step. This is a powerful primitive that enables us to design efficient algorithms. We refer to our model as the *known-delay* model.

To measure the time complexity of an algorithm in the known-delay model, we account for the *step complexity* that measures the number of times a process accesses shared registers, along with the *explicit-delay complexity* that is the sum of the explicit delays executed using the *delay* statement. Apart from the usual worst case complexity that indicates

*A preliminary version of this paper appeared in *Proceedings of 13th IEEE Symposium on Real-Time Systems* (RTSS 1992), pp. 12-21, and the full version appeared in *Distributed Computing*, Vol. 10(1), pp. 1-10, 1996.

†Bell Laboratories, 600 Mountain Avenue, Murray Hill, NJ 07974. alur@research.att.com.

‡The Open University, 16 Klausner street, P.O. Box 39328, Tel-Aviv 61392, Israel, and AT&T Bell Laboratories. gadi@research.att.com.

the maximum time it takes a process to attain its goal, we will also be interested in the *contention-free* complexity, which gives an upper bound on the time required for a process to attain its goal, when the process runs by itself without any interference from other processes. Since contention should be rare in well-designed systems, it is important to design algorithms that perform well also in the absence of contention. This was first pointed out in [11] where a mutual exclusion algorithm is presented, in which a process accesses shared registers only a constant number of times to enter its critical section in the absence of contention. A *fast* algorithm is, then, an algorithm with constant contention-free step complexity and zero contention-free explicit-delay complexity.

Mutual exclusion

The mutual exclusion problem is to design a protocol that guarantees mutually exclusive access, in a deadlock-free manner, to a critical section among a number of competing processes [8]. It is known that every solution to deadlock-free mutual exclusion among n processes in an asynchronous model must use n shared registers [6]. In fact, n registers are required even when there is an upper bound on time required for a single access, but this upper bound is *not* known a priori [13]. Fischer [11] showed that this lower bound is not applicable to the known-delay model by presenting a simple and elegant solution that uses a single (multi-writer) shared register. In his solution, a process needs to access the shared memory four times, and also needs to delay itself for Δ time units before it can enter its critical section. The disadvantage of this solution is that the contention-free explicit-delay complexity is also Δ.

The first fast solution was presented by Lamport [11]. In his solution, the number of times a process accesses the shared memory before entering its critical section in the absence of contention is constant — only five. Lamport's algorithm is used, for example, when the transaction processing system TUXEDO [22] [1] that provides programmers with a framework for building on-line transaction processing application in a distributed computing environment, is run on machines – like the MIPS 3000 series processors – that do not support an atomic test-and-set operation [14]. Lamport's algorithm provides fast access in the absence of contention, but, in the presence of contention, however small, the winning process may have to check the status of all other n processes. For the asynchronous model, it is known that in any mutual exclusion for two or more processes, there is no bound on the number of steps taken by the winning process; that is, in presence of contention the adversary can schedule the contending processes in such a way that each one of them will have to busy-wait [3].

Fischer's and Lamport's algorithms raise the question whether there is an algorithm where:

1. in the absence of contention, as in Lamport's algorithm, a process can always enter (and exit) its critical section fast without having to delay itself (i.e. constant step complexity and zero explicit-delay), and
2. in the presence of contention, as in Fischer's algorithm, a process does not have to check the status of all other n processes, but may need to delay itself (i.e. constant step-complexity and $O(\Delta)$ explicit-delay).

[1] TUXEDO and UNIX are registered trademarks of UNIX System Laboratories.

In Section 3, we present such an algorithm along with its proof of correctness.

Consensus

In the *consensus* problem, processes need to agree on a common output in the presence of possible failures [20]. For asynchronous systems, even when one process can fail, the consensus problem is not solvable [9, 12]. In Section 4, we show that in the known-delay model, there is a simple algorithm that tolerates any number of failures. Our algorithm is fast: in absence of contention, a process can decide after a constant number of steps without using the *delay* statement. In the worst case, the step-complexity is constant, and the explicit-delay complexity is Δ. Since consensus is universal [10], our results imply that atomic registers are universal in the known-delay model.

Related work

As mentioned earlier, our mutual exclusion algorithm is based on Fischer's timing-based solution and Lamport's fast algorithm [11]. Lamport's fast algorithm has been improved in the asynchronous model, so that the winning process, in the presence of contention, needs to check the status of only the currently contending processes (instead of all n processes) [7, 21].

Lamport's paper also contains a fast timing-based algorithm, like our algorithm, with constant worst case step complexity and $O(\Delta)$ worst case explicit delay. However, Δ used in his algorithm is not only an upper bound on the time required to perform an individual operation such as a memory access, but also on the time needed to execute the critical section. As he pointed out, in most situations, an algorithm that does not require this bound is needed. Our algorithm does not assume any bound on the time needed to execute the critical section.

In [13], the mutual exclusion problem is considered in a timing-based model similar to ours. An algorithm is presented that guarantees mutual exclusion even when run asynchronously, and also avoids deadlock when the timing constraints on the speeds of the processes are met. However, their algorithm does not provide fast access in the absence of contention. It is straightforward to combine the ideas of our algorithm and the algorithm of [13] to obtain an algorithm for mutual exclusion that provides fast access in absence of contention, guarantees mutual exclusion even when run asynchronously, and avoids deadlocks when the timing constraints are satisfied. Less related is the work in [5], where a variant of the mutual exclusion problem is studied in a message passing model with inaccurate clocks, where some bounds on the speeds of the processes and the time for message delivery are known.

Recently, the work in this paper has been extended to the *unknown-delay* model, where an upper bound exists on the memory-access time, but is not known a priori [1].

2 A timing-based model

In this section, we outline our model of distributed systems. Processes are modeled as (possibly infinite) state-machines communicating via shared memory that consists of atomic registers.

A (global) *state* of the system includes the state of each process and the values of all shared registers. An *event* is a single step of some process, and is either a read of a shared register, or a write to a shared register, or simply an update of the internal state of a process. Processes can also execute a delay statement $delay(d)$, for a positive integer d, and its effect on a state is similar to that of a skip statement.

As in the standard interleaving semantics, an *execution* $\overline{s}$ of the system is a finite or infinite alternating sequence $s_0 \xrightarrow{e_0} s_1 \xrightarrow{e_1} \cdots$ of states s_i and events e_i such that (1) the initial state s_0 satisfies the initial condition, and (2) every state s_{i+1} is derived from the previous state s_i by applying the event e_i in an appropriate manner. Each event belongs to a unique process. An index j is said to be the *successor* of an index i, for $j > i$, if both events e_i and e_j belong to the same process p, and no intermediate event e_k, for $i < k < j$, belongs to p.

In mutual exclusion, we assume that no processes fail (instead of allowing a process to terminate in its remainder region, we require that a process continues to take idling steps in its remainder region), however, in consensus, failures are possible. We consider only crash failures: a failed process simply ceases to participate. Also, we assume that all maximal executions are infinite, and thus, failures are detected only in infinite executions (technically, this implies that all processes do not fail). Formally, a process p is *nonfaulty* in an execution $\overline{s}$ iff either $\overline{s}$ is finite or infinitely many events in $\overline{s}$ belong to p. Notice that this definition assumes that every nonfaulty process keeps on taking steps. To accommodate this assumption in a problem, such as consensus, where a process can terminate successfully, we (implicitly) add a self-loop with a *skip* statement after the termination.

The notion of an execution captures only the asynchronous part of the system and not its timing requirements. Define the *explicit delay* of an event e, denoted by $d(e)$, to be d if e is the delay statement $delay(d)$, and 0 otherwise. We assume that each access to memory takes at most Δ time units. A *timed execution* of a system is a pair $(\overline{s}, \overline{t})$ such that $\overline{s}$ is an execution and $\overline{t}$ is a mapping that assigns a real-valued occurrence time t_i to each event e_i in $\overline{s}$ such that

1. the occurrence times are nondecreasing,
2. if $\overline{s}$ is infinite then the sequence of occurrence times is unbounded[2], and
3. if step j is the successor of step i then $d(e_i) < t_j - t_i \leq \Delta + d(e_i)$.

The last requirement captures the assumption regarding the bounds on execution speeds. A delay statement $delay(d)$ by a process p delays p for at least d and at most $\Delta + d$ time units before it can continue. For other statements, we simply require that a step of a process takes nonzero time with upper bound of Δ (note that adjacent events belonging to different processes can be assigned the same time). An execution $\overline{s}$ is said to be *timing consistent* iff there is some time assignment $\overline{t}$ such that $(\overline{s}, \overline{t})$ is a timed execution. Observe that our definition does not place any bounds on the time when a process takes its first step.

A problem such as consensus or mutual exclusion is usually specified by listing the properties to be satisfied by all the executions. An algorithm A satisfies a property ϕ in the asynchronous model iff all of its executions satisfy ϕ. Similarly, we say that an algorithm A satisfies ϕ in the known-delay model iff all the timing-consistent executions of A satisfy ϕ.

[2] Our results do not depend on this second requirement.

We consider two measures of time complexity: the *step complexity* and the *explicit-delay complexity*. For an execution $\overline{s}$ and indices $i \leq j$, the execution fragment $\overline{s}_{ij}$ denotes the subsequence $s_i \xrightarrow{e_i} \cdots \xrightarrow{e_{j-1}} s_j$ of $\overline{s}$. The step complexity of a process p in an execution fragment $\overline{s}_{ij}$ is the number of events in $\overline{s}_{ij}$ that belong to p, and the explicit-delay complexity of p in $\overline{s}_{ij}$ is the sum of explicit delays executed by p in $\overline{s}_{ij}$ (i.e. sum of $d(e_k)$ such that $i \leq k \leq j$ and e_k belongs to p). For each problem, worst-case and contention-free step complexity will be defined in an appropriate way.

As pointed out in [18], the definition of step complexity does not always capture the real time complexity of an algorithm. If one process is in a loop waiting for another process to complete some action, then increasing the execution speed of the first process increases its number of steps, without necessarily degrading the total system performance. This suggests that time complexity should be defined using timed executions. If the step complexity of a process p in an execution fragment $\overline{s}_{ij}$ is n and its explicit-delay complexity is d, then p takes at least d and at most $n\Delta + d$ time units to execute the fragment $\overline{s}_{ij}$. Consequently, upper bounds on the step and explicit-delay complexities give a (not necessarily tight) upper bound on the actual time complexity (for instance, system response time defined in [7]). For our investigation, it is enough to notice that all our upper bounds on the step and explicit-delay complexities are constants, and thus, imply a constant upper bound on the actual time complexity.

3 Mutual exclusion

3.1 Problem definition

The mutual exclusion problem is to design a protocol that guarantees mutually exclusive access to a critical section among a number of competing processes [8]. A solution to the problem should satisfy the following two properties,

- **Mutual exclusion:** No two processes are in their critical sections at the same time.
- **Deadlock freedom:** If some process p is in its entry code, then eventually some process (possibly different from p) is in its critical section, and if process p is in its exit code, then eventually p is in its remainder region.

While deadlock-freedom is essential, starvation-freedom which guarantees that any process that is trying to enter its critical section, eventually is in its critical section, is not as important in well-designed systems where contention is rare. When contention is rare it is important to design algorithms with low contention-free complexity.

We assume that each of the potentially n contending processes has a unique identifier taken from the set $\{1, ..., n\}$. An algorithm for mutual exclusion specifies entry code to be executed before entering the critical section, and exit code to be executed after leaving the critical section. Initially, each process is in its remainder region, and starts executing the entry code when the resource is required. A process returns to its remainder region after executing the exit code. To simplify the definitions of our complexity measures, we will assume that a process does not take steps in its critical section, and can take idling steps in its remainder region. It is assumed that no processes fail. Thus, a process is required to take infinitely many steps in an infinite execution. Note that a process need not participate in the mutual exclusion protocol, and can simply take idling steps in its remainder region.

The worst case step (or explicit-delay) complexity of the entry code of an algorithm is the maximum step (or explicit-delay) complexity of a process p in an execution fragment $\overline{s}_{ij} = s_i \stackrel{e_i}{\to} \cdots \stackrel{e_{j-1}}{\to} s_j$ such that

1. process p is in its entry code in state s_i, and is in its critical section in state s_j, and

2. no process is in its exit code or critical section in states s_k for $i \leq k < j$.

The second condition ensures that we start counting the number of steps of the winning process p only after the processes previously in the critical section have finished their exit codes. The worst case step (or explicit-delay) complexity of the exit code of an algorithm is the maximum step (or explicit-delay) complexity of a process p in an execution fragment $\overline{s}_{ij}$ such that process p is in its critical section in state s_i, and in its exit code in state s_j. The worst case step (or explicit-delay) complexity of an algorithm is the sum of the worst case step (or explicit-delay) complexity of its entry code and exit code.

The contention-free step (or explicit-delay) complexity of an algorithm is the maximum step (or explicit-delay) complexity of a process p in an execution fragment $\overline{s}_{ij}$ such that

1. process p is in its entry code in state s_i, and is in its exit code in state s_j,

2. in all states s_k, $i \leq k \leq j$, process p is not in its remainder region, and all other processes $p' \neq p$ are in their remainder regions.

An algorithm for mutual exclusion is said to be *fast* if it has constant contention-free step complexity, and zero contention-free explicit-delay complexity. For a fast algorithm, the time taken by a process to enter its critical section is independent of the total number of processes, or the upper bound on the speeds of all the processes.

3.2 Previous solutions

Our algorithm combines the features of Lamport's fast algorithm, and Fischer's timing-based algorithm. We present both these algorithms here, and refer the reader to [11] for correctness proofs.

Figure 1 presents Lamport's algorithm. Let us say that a process p *enters the critical section along path* α if it finds $x = p$ before the entry to the critical section, and p *enters the critical section along path* β if it reads $y = p$ before the entry to the critical section. Process p first sets its flag $b[p]$ to declare its request. It sets the multi-writer register x to p, and then checks the value of y. When it finds $y = 0$, it sets y to p and then checks the value of x. If it finds $x = p$ at this step, then it can enter the critical section along path α. It should be clear that at most one process can enter its critical section along path α. If a process finds $x \neq p$ then it waits till all the flags $b[q]$ are reset. Note that after a process executes the **for** loop, the value of y remains unchanged until some process leaving its critical section resets it to 0. Thus, there can be at most one process p that can find $y = p$, and this process gets to enter its critical section along path β. Notice that, in the contention-free case, a process always uses path α, resulting in fast access. However, in the

Shared registers: x, y: integer, $b[1..n]$: boolean array.
Initially: $y = 0$, $b[p] = \mathit{false}$ for all p.

```
start: b[p] := true;
       x := p;
       if y ≠ 0 then b[p] := false;
                     await (y = 0);
                     goto start fi;
       y := p;
       if x ≠ p then b[p] := false;
                     for q := 1 to n do await (¬b[q]) od;
                     if y ≠ p then await (y = 0);
                                   goto start fi fi;
       critical section;
       y := 0;
       b[p] := false;
```

Figure 1: Lamport's Fast Algorithm – process p's program.

Shared register: y: integer
Initially: $y = 0$.

```
repeat  await (y = 0);
        y := p;
        delay(Δ);
until   y = p;
critical section;
y := 0;
```

Figure 2: Fischer's Timing-based Algorithm – process p's program.

presence of contention, however small, the winning process may have to follow path β that involves checking the status of all other n processes.

Fischer's timing-based algorithm is presented in Figure 2. In this algorithm the *delay* statement ensures that after a process finishes the delay statement, the value of y remains unchanged until some process leaving its critical section resets it to 0. A process needs to access the shared memory at least four times and needs to delay itself for Δ time units before it can enter its critical section. The disadvantage of this solution is that even in the absence of contention a process needs to delay itself.

3.3 Fast timing-based algorithm

Our new timing-based algorithm is shown in Figure 3. The mutual exclusion property crucially depends on the assumption about the speeds of the processes. The algorithm is

```
Shared registers: x, y: integer; z: boolean
Initially: y = 0, z = false.
start: x := p;
       await (y = 0);
       y := p;
       if x ≠ p then delay(2·Δ);
                     if y ≠ p then goto start fi;
                     await (¬z)
       else z := true;
       critical section;
       z := false;
       if y = p then y := 0 fi;
```

Figure 3: Fast Timing-based Algorithm – process p's program.

fast: the contention-free step complexity is 8 (5 steps in the entry code and 3 steps in the exit code), and the contention-free explicit-delay complexity is 0.

Let us say that a process p *enters the critical section along path* α if it executes the assignment $z := true$ before the entry to the critical section, and p *enters the critical section along path* β if it executes the statement **await** $(\neg z)$ before the entry to the critical section. As in Lamport's algorithm, the sequence write(x), read(y), write(y), read(x) is used for the fast access along path α. If a process finds $x \neq p$ then it attempts to enter along path β. However, instead of checking the status of all n processes as in Lamport's algorithm, it delays itself. A process p enters its critical section along path β if it finds $y = p$ after the delay. The *delay* statement plays two roles:

1. The delay ensures that if process p finds $y = p$ after the delay statement, no other process can change the value of y until process p sets y to zero in its exit code, and hence it follows that at most one process can enter along path β. To see this we observe that, as in Fischer's protocol, the *delay* statement ensures that, while process p delays itself, every other process q that got past the $await(y = 0)$ statement finishes the assignment $y := q$. Furthermore, while process p delays itself, a process in the exit code, that got past the condition in the *if* statement finishes the assignment $y := 0$.

2. Along path β, the delay statement replaces the loop that checks n registers. However, unlike in Lamport's case, a process entering its critical section along path β does not know whether or not some other process entered its critical section along path α. An additional boolean flag z, initially false, is used for this purpose. A process entering its critical section along path α sets z, and a process entering its critical section along path β must wait until z is reset before entering. Here the second role of the delay statement becomes important. It ensures that the process entering its critical section along path α sets z *before* a process along path β gets a chance to test z. To check this observe that a process q entering its critical section along path α executes 3 steps between the $await(y = 0)$ statement and the entry to its critical section. If process q is before the assignment $y := q$ when process p reaches its delay statement, then process p will find $y \neq p$ after it finishes the delay. If process q is after the assignment $y := q$ when process p reaches its delay statement, then

since process p delays itself by time $2 \cdot \Delta$, it will find $z = true$ after the delay, unless process q leaves its critical section in the meanwhile.

The exit code needs to reset both the locks y and z to ensure guaranteed progress. This is tricky: there may be a process waiting at $await(\neg z)$ that can enter its critical section immediately after z is reset, and there may be a process waiting at $await(y = 0)$ that can enter its critical section along path α immediately after y is reset. It seems that doing the two reset actions in any order leads to a possible violation of the mutual exclusion requirement. However, observe that if process p is in its critical section, and y is different from p, then process p must have entered along path α, and there is some other contending process that will eventually win along path β, so in this case there is no need to reset y. On the other hand, if y equals p then there is no process waiting at the $await(\neg z)$ statement, and hence y can be safely reset.

The properties of the algorithm are summarized in the following theorem; the proof of correctness appears in the next section.

Theorem 1 *All timing consistent executions of the algorithm of Figure 3 satisfy mutual exclusion and deadlock freedom.*

The various complexity measures for the algorithm are analyzed in Section 3.5. The algorithm has also been mechanically verified for $n = 3$ processes using the verification tool COSPAN (see [2] for details). Note that the algorithm fails to guarantee mutual exclusion if the timing constraints are not met. That is, only the timing consistent executions satisfy mutual exclusion. We point out that the statement $delay(2 \cdot \Delta)$ can be replaced by two $delay(\Delta)$ statements; the first is placed at the same place as before and the second is placed before the $await(\neg z)$ statement.

3.4 Proof of correctness

A *state* of the algorithm is completely described by the values of the shared registers x, y, and z, and the values of the location counters of all the processes. We will use ℓ_p to denote the location counter of process p. As shown in Figure 4 (its annotations will be explained later), ℓ_p ranges over the set 0..11. Notice that the delay statement has been split into two consecutive delay statements. Consider an infinite execution

$$\overline{s} \;=\; s_0 \xrightarrow{e_1} s_1 \xrightarrow{e_2} \cdots$$

of the algorithm. In the initial state s_0, all the registers, except for x, have the value 0. The initial value of x is of no importance. Each state s_{i+1} is obtained from its previous state s_i by applying the event e_i. A process is in its critical section when its location counter is 9; the location counter changes to 10, when it executes the assignment $z := false$. Also note that when the control of process p is at location 6, it moves to location 0 if $y \neq p$. It is not necessary to model the remainder region separately to prove correctness. Since executions are failure-free, and we consider only infinite executions, it follows that every index has a successor.

We want to prove correctness for all timing consistent executions. The following lemma captures the crucial aspect of the assumptions about the speeds of various processes. It

start (0) $x := p$;
(1) **await** $(y = 0)$;
(2) $y := p$;
(3) $[\ x = p \rightarrow (\ y \neq 0\ \wedge\ \forall q.\ (\ell_q \in 0..5 \vee (\ \ell_q \in \{6,10\} \wedge y \neq q)))]$
if $x \neq p$ **then** (4) **delay**(Δ);
(5) $[\ y = p \rightarrow \forall q.\ \ell_q \notin \{2,3,7,11\}\ ]$
delay(Δ);
(6) $[\ y = p \rightarrow \forall q.\ (\ \ell_q \notin \{2,3,7,8,11\} \wedge (z \vee \ell_q \neq 9))]$
if $y \neq p$ **then goto** *start* **fi**;
(7) $[\ y = p \wedge \forall q \neq p.\ (\ \ell_q \notin \{2,3,7,8,11\} \wedge (\ z \vee \ell_q \neq 9))]$
await $(\neg z)$
else (8) $[y \neq 0 \wedge \forall q \neq p. (\ell_q \in 0..1 \vee (x \neq q \wedge (\ell_q \in 2..5 \vee (\ell_q \in \{6,10\} \wedge y \neq q))))]$
$z := true$;

(9) $[\ [z \wedge y \neq 0 \wedge \forall q \neq p.\ (\ell_q \in 0..1 \vee (x \neq q \wedge \ell_q \in 2..7) \vee (y \neq q \wedge \ell_q = 10))\ ] \vee$
$[\neg z \wedge y = p \wedge \forall q \neq p.\ \ell_q \in \{0,1,4..6,10\}]\ ]$
critical section;

$z := false$;
(10)$[\ y = p \rightarrow \forall q.\ (\ \ell_q \neq 11 \wedge (\ \ell_q \notin 2..3 \vee x \neq q\ ))]$
if $y = p$ **then**
(11) $[\ y \neq 0\ \wedge\ \forall q.\ (\ \ell_q \neq 11 \wedge (\ \ell_q \notin 2..3 \vee x \neq q\ ))]$
$y := 0$ **fi**;

Figure 4 : Annotations for Process p's Program.

says that by the time process p executes one of the *delay* statements, every other process q at one of the locations 2, 3, 8, or 11 executes at least one statement. This property of timing constraints suffices to prove correctness.

Lemma 1 *Let $\overline{s}$ be a timing-consistent execution, p be a process, and e_i and e_j be events of process p such that j is the successor of i. If ℓ_p is in 4..5 in state s_i (see Figure 4), then for every process q, it cannot be the case that ℓ_q is in $\{2,3,8,11\}$ in state s_i, and e_k does not belong to q for all $i \leq k < j$.*

Proof: Consider a timing-consistent execution $\overline{s}$, and let $(\overline{s}, \overline{t})$ be a timed execution. Let p be a process. Let e_i and e_j be events of process p such that e_j is the successor of e_i, and ℓ_p is in 4..5 in state s_i. Since $d(e_i) = \Delta$, it follows that $t_j - t_i > \Delta$. By contradiction, assume that q is a process such that ℓ_q is in $\{2,3,8,11\}$ in state s_i, and e_k does not belong to q for all $i \leq k < j$. Let k be the greatest index smaller than i such that e_k belongs to q. In state s_k, ℓ_q is in $\{2,3,8,11\}$, and hence, $d(e_k) = 0$. Let k' be the successor of k (such a successor exists, since $\overline{s}$ has infinitely many steps by each process). Then $k' > j$. This implies $t_{k'} - t_k \geq t_j - t_i > \Delta$; a contradiction to the timing constraint $t_{k'} - t_k \leq \Delta$. ∎

3.4.1 Mutual exclusion

Mutual exclusion requirement is specified by the following safety formula [3]:

$$\phi_{me} \;=\; \forall\, p, q, p \neq q.\ \Box\, \neg\ (\ \ell_p = 9 \wedge \ell_q = 9\ ).$$

To prove this property we show that the following formula Φ is a global invariant of the algorithm (i.e., Φ holds in every state appearing on a timing-consistent execution of the algorithm):

$$\Phi = \forall\, p.\ \bigvee_{m \in 0..11} (\ell_p = m \wedge \phi^p_m),$$

that is, for every process p and location m, whenever process p is at location m, the formula ϕ^p_m holds. The formula ϕ^p_m is an assertion associated with the location m of process p. These assertions are shown in the annotated algorithm of process p of Figure 4 in square brackets following each location m. There is no annotation for locations 0, 1, 2, and 4 and for these locations the corresponding assertion is the formula *true*.

First we prove that

$$\Box\, \Phi\ \rightarrow\ \phi_{me}.$$

Suppose not. Then there exist two distinct processes p, q, and a state s_i of some execution such that in which Φ is true, $\ell_p = 9$ and $\ell_q = 9$. This implies ϕ^p_9, which contradicts $\ell_q = 9$ (the reader can verify that $\phi^p_9 \wedge \ell_q = 9$ is logically equivalent to false). Hence, to prove mutual exclusion it suffices to show that Φ is indeed a global invariant.

Consider a execution $\overline{s}$. We will prove that Φ holds in every state s_i by induction on i. Observe that Φ is trivially true in the initial state s_0 since all location registers are initially 0, and the corresponding assertions ϕ^p_0 are defined to be *true* (in fact, the initial values of the shared registers x, y, z are irrelevant for proving Φ). Now assume that Φ holds in all states s_0 through s_i, and we will show that it continues to hold in state s_{i+1} obtained by executing one step of process p (that is, step e_i belongs to process p). This is done by a case analysis on the value of location ℓ_p in state s_i. Since this analysis is straightforward, we give detailed proofs only for 2 illustrative and relatively difficult cases:

Case $\ell_p = 2$

State s_{i+1} is obtained by executing the assignment statement $y := p$ by process p. Since ℓ_p is 3 in s_{i+1}, we need to show that (1) ϕ^p_3 holds in s_{i+1}, and (2) for every other process q and every location $m \in 0..11$, the truth of the conjunct $(\ell_q = m \wedge \phi^q_m)$ is unaffected by this step. These two conditions are similar to the requirements of sequential correctness and interference freedom, respectively [16, 17]. First let us prove ϕ^p_3:

$$\phi^p_3 \;=\; [\ x = p \rightarrow (\ y \neq 0\ \wedge\ \forall q.\ (\ \ell_q \in 0..5 \vee (\ \ell_q \in \{6, 10\} \wedge y \neq q\ )))].$$

Suppose x equals p in s_{i+1}. Clearly, x equals p in s_i also. The value of y in s_{i+1} is p, and so the conjunct $y \neq 0$ is true. Consider another process $q \neq p$. We need to show that either $\ell_q \in 0..5$ or $\ell_q \in \{6, 10\}$ with $y \neq q$. Since y equals p, we need to show that $\ell_q \in \{0..6, 10\}$. Note that the value of ℓ_q is same in s_{i+1} as in s_i, and in state s_i the invariant

[3] See appendix for a brief introduction to temporal logic.

Φ holds. Now the observation that for each location $m \in \{7..9, 11\}$, the formula ϕ^q_m implies $(x \neq p \vee \ell_p \neq 2)$ shows that $\ell_q \in \{0..6, 10\}$ in state s_i. This completes the proof that ϕ^p_3 holds in s_{i+1}. We also need to prove interference freedom. Consider another process $q \neq p$. It suffices to show that for every location $m \in 0..11$, ϕ^q_m holds in state s_{i+1} assuming that ϕ^q_m and ϕ^p_2 hold in state s_i. This is completely straightforward.

Case $\ell_p = 4$

Let us consider the case when ℓ_p changes from 4 to 5. Recall that the *delay* statement has no effect on other state registers. First we need to show that ϕ^p_5 holds in state s_{i+1}.

$$\phi^p_5 \ = \ [\ y = p \rightarrow \forall q.\ \ell_q \notin \{2, 3, 7, 11\}\ ].$$

Assume y equals p in state s_{i+1} (and, hence in state s_i also), and let $q \neq p$ be another process. Note that the value of ℓ_q is the same in states s_i and s_{i+1}. We will show that ℓ_q cannot be either 2, 3, 7, or 11 in state s_i.

Since Φ holds in state s_i, and ϕ^q_7 implies y equals q, but y equals p in s_i. Hence, ℓ_q cannot be 7 in s_i.

Next we prove that $\ell_q \notin \{2, 3, 11\}$ in state s_i. Let us assume to the contrary. Let $j \leq i$ be the greatest index such that ℓ_p equals 4 in all states s_k for $j \leq k \leq i$ (i.e., process p reached the *delay* statement at j-th step of $\overline{s}$). Since no other process can change the value of y to p, by a backward induction from i to j, we can show that y equals p in all states s_k for $j \leq k \leq i$. We have assumed that $\ell_q \in \{2, 3, 11\}$ in state s_i. Observe that the control of process q can move from location 1 to location 2 only if $y = 0$, and from location 10 to location 11 only if $y = q$, and if the control moves from location 2 to location 3 then the value of y becomes q. Since $y = p$ holds in all states s_k for $j \leq k \leq i$, it follows that ℓ_q is unchanged in all states s_k for $j \leq k \leq i$. This means that process q did not take a step while process p executed its delay statement. Hence, the execution $\overline{s}$ does not satisfy the timing constraint, a contradiction, and this proves that ϕ^p_5 holds in state s_{i+1}. Proving interference freedom is particularly easy: the fact that ℓ_p changes from 4 to 5 leaves the truth of all the subformulas in all ϕ^q_m's unchanged for processes $q \neq p$ and all locations $m \in 0..11$.

Note that, unlike the standard rules for proving invariance [15], we cannot show that in all states s, if Φ holds in s then Φ continues to hold when process p moves from location 4 to location 5. We need to reason about the execution that leads to s, and to prove that Φ continues to hold, we need the fact that Φ holds in all previous states of the execution, and that the execution satisfies the timing constraint.

3.4.2 Deadlock freedom

The deadlock freedom property is specified by the conjunction $\phi^0_{df} \wedge \phi^1_{df}$, where

$$\phi^0_{df} = \ \forall p.\ \Box\ [\ \ell_p \in 1..8 \rightarrow \exists q.\ \Diamond\ \ell_q = 9\ ]$$

says that if a process p is in its entry code then some process eventually is in the critical section, and

$$\phi^1_{df} = \ \forall p.\ \Box\ [\ \ell_p \in 10..11 \rightarrow \ \Diamond\ \ell_p = 0\ ]$$

says that if a process p is in its exit code then it eventually is in its remainder region (corresponding to location 0). The exit code is a straight-line code, and hence ϕ^1_{df} is immediate. The proof of ϕ^0_{df} depends upon two additional safety properties:

$$\phi_y \;=\; \forall p.\, \Box\; (\; y = p \rightarrow \ell_p \notin 0..1\;),$$

and

$$\phi_z \;=\; \Box\; (\; z \rightarrow \exists p.\; \ell_p = 9\;).$$

Both the properties can be shown to be invariants of the algorithm.

Consider a execution $\overline{s} = s_0, s_1, \ldots$ of the algorithm. We will prove that the property ϕ^0_{df} holds for this execution. The proof is by contradiction. Suppose it does not. The following sequence of assertions leads to the desired contradiction. In the following sequence, each assertion is followed by a brief justification.

1. There exists i such that $\ell_p \in 1..8$ in s_i for some p, and $\ell_q \neq 9$ for all q in all states s_j, $j \geq i$.

   This is because the execution $\overline{s}$ violates ϕ^0_{df} by assumption.

2. There exists j such that $\ell_p \in 1..8$ in s_j for some process p, and in all states s_k with $k \geq j$, $z = \mathit{false}$ and $\ell_q \notin 9..11$ for all processes q.

   Since the execution is fair, starting from state s_i every process eventually leaves the locations 10..11, and from this, and from ϕ_z, assertion 2 follows.

3. If a process q is trying in state s_k (i.e., $\ell_q \in 1..8$) for $k \geq j$, then eventually $\ell_q = 1$.

   This follows from the facts that process q keeps taking steps, it never enters its critical section, and by assertion 2, cannot get blocked at location 7 (since z stays *false*).

4. There is some state s_k, $k \geq j$, with $y \neq 0$.

   The trying process of assertion 2 will eventually reach location 1 according to assertion 3, and then, if y stays 0 it will eventually execute the assignment at location 2.

5. In all states s_l with $l \geq k$, y is nonzero.

   Following the state s_k nobody is ever at location 11, which is the only place where y is reset.

6. There exists $l \geq k$ and process q such that $y = q$ in all states s_m with $m \geq l$.

   From assertion 5, it follows that for process q, if $\ell_q \neq 2$ in s_k, then in all the following states also $\ell_q \neq 2$ (no process can get past the *await* statement at location 1). Consequently, after all the processes at location 2 in state s_k execute the assignment, the value of y cannot change.

7. From assertions 6 and 3, there exists $m \geq l$ such that $y = q$ and $\ell_q = 1$ in the state s_m. This is a contradiction to the global invariant ϕ_y.

3.5 Complexity

We now examine the performance of our algorithm using the complexity measures defined earlier.

Theorem 2 *For the algorithm of Figure 3,*

1. *the contention-free step complexity is 8,*
2. *the contention-free explicit-delay complexity is 0,*
3. *the worst case step complexity is 13, and*
4. *the worst case explicit-delay complexity is* $4 \cdot \Delta$.

Proof: First we point out that for proving the theorem, there is no need to use the fact that the algorithm is deadlock free or that it satisfies mutual exclusion.

The property of fast access in absence of contention is obvious: in absence of contention, process p executes 3 write statements and 2 read statements before entering its critical section along path α, and executes 2 write statements and 1 read statement in its exit code; a total of 8 steps. In that case, no explicit delay statements are executed. This proves part 1 and part 2 of the theorem.

For the worst case complexity, we first observe that while executing its entry code, a process may busy-wait (loop) in three ways:

1. waiting for y to be set to zero in the statement **await** $(y = 0)$,
2. waiting for z to be set to *false*, in the statement **await** $(\neg z)$.
3. executing the "**goto** *start*" statement several times.

In the first two cases, once a process is busy waiting it may continue only when some other process *in its exit code* changes the values of either y or z. Similarly, in any interval in which some process executes the the "**goto** *start*" statement twice, some other process must have set the values of y to zero in its exit code.

This implies that, for any given process p, if we look at an execution fragment $\overline{s}_{ij} = s_i \xrightarrow{e_i} \cdots \xrightarrow{e_{j-1}} s_j$ where,

1. process p is in its entry code in state s_i, and is in its critical section in state s_j,
2. no process is in its exit code or critical section in states s_k for $i \leq k < j$,

we get that in $\overline{s}_{ij}$, process p may have executed at most once, the two await statements and the goto statement. Thus, to find out what is the maximum number of steps that p can take in $\overline{s}_{ij}$, we simply have to count the number of steps p can take in its entry code assuming no waiting.

Now, when counting steps, a winning process takes at most 7 steps from *start* to its critical section, and delays itself by $2 \cdot \Delta$ (by definition, the delay statement contributes 1 to the step complexity, and $2 \cdot \Delta$ to the explicit-delay complexity).

When a process leaves the exit code, the next winning process may be either at *start*, or at one of the two *await* statements, or executing the delay, or just after setting y to its identifier. In the last case, it may need to take two steps to reach the beginning of the entry code, and will also need to execute the delay statement (which contributes one additional step), to reach the beginning of the entry code.

Finally, we observe that a process executes at most 3 steps in the exit code without any delay. Thus, the overall worst case step complexity is 13, and worst case explicit-delay complexity is $4 \cdot \Delta$. ∎

4 Consensus

4.1 Problem definition

The *consensus problem* is to design an algorithm in which all correct processes reach a common decision based on their initial inputs [20]. Formally the problem is defined as follows. There are an arbitrary number of processes, we do not require unique identifiers. Each process has an input value chosen from $\{0, 1\}$. A process *decides* on a value $v \in \{0, 1\}$ by executing the statement *decide*(v). It may decide at most once. The consensus problem requires that

- **Agreement:** If one process decides on a value v, and another on value v', then $v = v'$.
- **Validity:** If a process decides on the value v then v equals the input value for some process.

Apart from the above safety requirements, we want the correct processes to eventually decide:

- **Wait freedom:** No process takes infinitely many steps without deciding.

The requirement of wait-freedom means that one process cannot prevent another process from reaching a decision, and thus the algorithm must tolerate arbitrary number of process failures. Wait freedom implies that each process either takes only finitely many steps, or decides on some value (we assume that a process takes infinitely many idling steps after it decides).

As in mutual exclusion, we consider the step and explicit-delay complexities in the worst and the contention-free cases. The worst case step (or explicit-delay) complexity of a consensus algorithm is the maximum step (or explicit-delay) complexity of a process p in an execution fragment $\overline{s}_{ij} = s_i \xrightarrow{e_i} \cdots \xrightarrow{e_{j-1}} s_j$ such that both the extreme events e_i and e_{j-1} belong to p. For the contention-free case, we take the maximum over execution fragments $\overline{s}_{ij}$ such that both e_i and e_{j-1} belong to p, and for every process q whose input is different from p, either q has decided in state s_i or q has not started in state s_j. Thus, in the contention-free case, there is no interference from processes with conflicting input while p executes; every process with conflicting input has either decided before p starts, or starts only after p finishes. As before, we want our solution to be fast.

Shared register: $y \in \{\perp, 0, 1\}$; $x[0..1]$: array of boolean
Initially: $x[0] = x[1] = \mathit{false}$; $y = \perp$

$x[v] := \mathit{true}$;
if $y = \perp$ **then** $y := v$ **fi**;
if $x[1-v]$ **then** $\mathit{delay}(\Delta)$ **fi**;
$\mathit{decide}(y)$;

Figure 5: Fast Timing-based Consensus Algorithm – program for process with input $v \in \{0, 1\}$.

4.2 Fast timing-based algorithm

The code to be executed by a process with input v is shown in Figure 5. For each input $v \in \{0, 1\}$, there is a flag $x[v]$, initially *false*. The register y is initially $\perp$, and contains the current decision value. A process with input v sets the flag $x[v]$, and if there is no current decision value, then executes $y := v$. The current decision value changes only if two processes with different inputs read $y = \perp$ before any assignment to y finishes. But in this case, both the flags $x[0]$ and $x[1]$ will be set, and every process is forced to execute the delay statement before deciding. The delay statement ensures, as in the case of Fischer's mutual exclusion algorithm, that the value of y will stay unchanged after some process finishes the delay.

The properties of the algorithm are summarized in the next theorem:

Theorem 3 *Algorithm of Figure 5 satisfies the properties of validity, agreement, and wait-freedom in the known-delay model.*

Proof: Observe that no process ever writes $\perp$ to y, and writes its input to y if it finds $y = \perp$. It follows that when a process decides, the value of y is in $\{0, 1\}$. To show validity, suppose all processes start with the same input, say, 0. Then $y \in \{\perp, 0\}$ is an invariant of the execution. It follows that no process ever decides on 1 in this case.

To prove agreement, consider a timed execution $(\overline{s}, \overline{t})$. Suppose that two processes decide on conflicting values in $\overline{s}$. This implies that there exists a process with input 0 that writes to y, as well as a process with input 1 that writes to y. Let i_0 be the least index such that a process with input 0 finds $y = \perp$ at step i_0, and i_1 be the least index such that a process with input 1 finds $y = \perp$ at step i_1. Since the flags $x[v]$ are set at the beginning, it follows that $x[0]$ is set in all states following s_{i_0} and $x[1]$ is set in all states following s_{i_1}. Consequently, in state s_i, with $i = \mathit{max}(i_0, i_1)$, both $x[0]$ and $x[1]$ are set, and $y = \perp$.

Let j be the smallest index such that some process writes to y at step j. Clearly, $y \neq \perp$ holds in all states $s_{j'}$ for $j' > j$. Hence, $j > i$, and every process that decides, executes the delay statement. Let p be the first process to decide, say, at step k.

We want to show that no process changes the value of y after step k, and this implies agreement. Assume to the contrary. Suppose q writes to y at step $k' > k$. Suppose that p checks the *if* clause of the delay statement at step m. It follows that $m > j$, and in state s_m, $y \neq \perp$. This means that q checks the value of y at some step $m' < m$. Since p executes the delay statement, $t_k - t_m > \Delta$. This implies $t_{k'} - t_{m'} \geq t_k - t_m > \Delta$, a contradiction to the upper bound Δ on the speed of q.

Each process executes a bounded number of steps, and there is no waiting. Wait-freedom follows trivially. ∎

The complexity of the algorithm is computed simply by counting the number of steps. In the contention-free case, a process follows the sequence write(x), read(y), write(y), read(x), before deciding. In the worst case, it may execute an additional delay statement.

Theorem 4 *For Algorithm of Figure 5, in the contention-free case, step complexity is 5, and the explicit-delay complexity is 0. In the worst case, step complexity is 6, and the explicit-delay complexity is Δ.*

Since consensus is universal [10], our results imply that atomic registers are universal in the known-delay model.

Acknowledgements

We thank Yehuda Afek, Hagit Attiya, Eli Gafni and Michael Merritt for helpful discussions, and the referees for detailed and useful suggestions.

References

[1] R. Alur, H. Attiya, and G. Taubenfeld. Time-adaptive algorithms for synchronization. *Proc. 26th ACM Symp. on Theory of Computing*, pp. 800–809, 1994. To appear in *SIAM Journal on Computing.*

[2] R. Alur, A. Itai, R. Kurshan, and M. Yannakakis. Timing verification by successive approximation. *Information and Computation*, 118(1):142–157, 1995.

[3] R. Alur and G. Taubenfeld. Results about fast mutual exclusion. In *Proc. 13th IEEE Real-Time Systems Symp*, pp. 12–21, 1992.

[4] R. Alur and G. Taubenfeld. How to share an object: A fast timing-based solution. In *Proc. of the 5th IEEE Symp. on Parallel and Distributed Processing*, pp. 470–477, 1993.

[5] H. Attiya and N. Lynch. Time bounds for real-time process control in the presence of timing uncertainty. *Proc. 10th Real-Time Systems Symp.*, pp. 268–284, 1989.

[6] J. Burns and A. Lynch. Bounds on shared memory for mutual exclusion. *Information and Computation*, 107:171–184, 1993.

[7] M. Choy and A. Singh. Adaptive solutions to the mutual exclusion problem. In *Proc. 12th ACM Symp. on Principles of Distributed Computing*, pp. 183–194, 1993.

[8] E.W. Dijkstra. Solution of a problem in concurrent programming control. *Communications of ACM*, 8(9), 1965.

[9] M. J. Fischer, N. A. Lynch, and M. S. Paterson. Impossibility of distributed consensus with one faulty process. *Journal of the ACM*, 32(2):374–382, 1985.

[10] M. Herlihy. Wait-free synchronization. *ACM Trans. on Programming Languages and Systems*, 11(1):124–149, 1991.

[11] L. Lamport. A fast mutual exclusion algorithm. *ACM Trans. on Computer Systems*, 5(1):1–11, 1987.

[12] C. M. Loui and H. Abu-Amara. Memory requirements for agreement among unreliable asynchronous processes. *Advances in Computing Research*, 4:163–183, 1987.

[13] N. Lynch and N. Shavit. Timing-based mutual exclusion. *Proceedings of the 13th IEEE Real-Time Systems Symposium*, pp. 2–11, 1992.

[14] M. R. MacBlane. Source level atomic test-and-set for the TUXEDO System source product. UNIX System Laboratories, May 14, 1991.

[15] Z. Manna and A. Pnueli. The temporal logic of reactive and concurrent systems. Springer-Verlag, 1991.

[16] S. Owicki and D. Gries. An axiomatic proof technique for parallel programs. *Acta Inf.*, 6(4), 1976.

[17] S. Owicki and L. Lamport. Proving liveness properties of concurrent programs. *ACM Transactions on Programming Languages and Systems*, 4(3), 1982.

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

[19] A. Pnueli. The temporal logic of programs. *Proc. 18th Symp. on Foundations of Computer Science*, 1977.

[20] M. Pease, R. Shostak, and L. Lamport. Reaching agreement in the presence of faults. *Journal of the ACM*, 27(2):228–234, 1980.

[21] E. Styer. Improving fast mutual exclusion. *Proc. 11th ACM Symp. on Principles of Distributed Computing,* pp. 159–168, 1992.

[22] *TUXEDO System Release 4.0 – Product Overview.* AT&T, 1990.

A Temporal logic

The use of temporal logic as a formalism for specifying the behavior of programs over time was first proposed by Pnueli [19]. Here, we provide a brief and informal introduction to (linear) temporal logic, see [15] for a detailed introduction.

The formulas of temporal logic are built from the state predicates (assertions about state registers) using first-order quantifiers, Boolean connectives such as $\wedge$ (meaning "and"), $\vee$ (meaning "or"), $\rightarrow$ (meaning "implies"), and temporal connectives such as $\Box$ (meaning "henceforth"), $\Diamond$ (meaning "eventually"), U (meaning "unless").

The formulas are interpreted over infinite state sequences, called *executions*. A execution $\overline{s}$ is an infinite sequence $s_0 s_1 \ldots$ of states s_i. Each state s_i provides an interpretation to all state registers. The semantics of the logic is defined using the satisfaction relation $(\overline{s}, i) \models \phi$, meaning ϕ holds in the i-th position of the execution $\overline{s}$. For a state predicate ϕ, $(\overline{s}, i) \models \phi$ iff ϕ evaluates to true in state s_i. The meaning of Boolean connectives and first-order quantifiers is as usual. Below we give the semantic definitions of the temporal connectives used in this paper:

- $(\overline{s}, i) \models \Box\phi$ iff $(\overline{s}, j) \models \phi$ for all $j \geq i$, that is, the formula $\Box\phi$ holds at position i of execution $\overline{s}$ iff starting from position i, ϕ holds in all later positions $j \geq i$.
- $(\overline{s}, i) \models \Diamond\phi$ iff $(\overline{s}, j) \models \phi$ for some $j \geq i$, that is, the formula $\Diamond\phi$ holds at position i of execution $\overline{s}$ iff starting from position i, ϕ holds eventually in some later position $j \geq i$.
- $(\overline{s}, i) \models \phi\, U\, \psi$ iff either $(\overline{s}, j) \models \phi$ for all $j \geq i$, or for some $j \geq i$, $(\overline{s}, j) \models \psi$ and $(\overline{s}, k) \models \phi$ for all $i \leq k < j$, that is, the formula $\phi\, U\, \psi$ holds at position i of execution $\overline{s}$ iff starting from position i, ϕ continues to hold until the first time ψ becomes true.

For instance, consider a execution $\overline{s} = s_0 s_1 \ldots$ of the algorithm of Figure 3, and the formula

$$\phi_{df} = \ \forall i.\ \Box\ [\ \ell_p \in 1..10 \rightarrow \exists j.\ \Diamond\ \ell_q = 11\ ].$$

Then ϕ_{df} holds in position k of $\overline{s}$ iff for every process i, and for every position $k' \geq k$, if state $s_{k'}$ satisfies the state predicate $\ell_p \in 1..10$, then there exists some process j, and some position $k'' \geq k'$ such that state $s_{k''}$ satisfies the state predicate $l_q = 11$.

A execution $\overline{s}$ satisfies a temporal logic formula ϕ iff $(\overline{s}, 0) \models \phi$, that is, ϕ holds in the initial position of $\overline{s}$. A program P is modeled by the set of all its possible executions. A program P satisfies a temporal logic specification ϕ iff every execution $\overline{s}$ of P satisfies ϕ.