ISSN 1883-3756

白鷗大学情報処理教育研究センター

第1号　2008年12月

年報

研　究　論　文

- Temporal Orders of Objects in the Shared Memory Model
- 心理データにおけるサンプルサイズの妥当性をめぐって
- 事業部長の利益業績評価指標と企業の資本利用効率との関係
- ウィルヘルム・シッカードの1623年のテュービンゲンの計算機
- 16進－10進変換課題実行時の事象関連電位を用いた学習進捗度計測の試み

公　開　講　座

研　究　会　報　告

公開講座報告

その他学会報告等

報　告　・　資　料

平成19年度事業報告

白鷗大学情報処理教育研究センター

研究論文

Temporal Orders of Objects in the Shared Memory Model

Yoshihide Igarashi
Professor Emeritus, Department of Computer Science, Gunma University
Tom Altman
Department of Computer Science and Engineering, University of Colorado at Denver

Abstract

We consider a set of users wishing to receive a service in an asynchronous distributed system. Users in the system declare their wishes and then wait to gain admittance to be served. For such systems, it is essential to decide the temporal order among these wishes. The mutual exclusion problem and its generalizations, such as the k-exclusion problem and the group mutual exclusion problem, cope with the process to determine the temporal order among objects occurred. It is not hard to design suitable algorithms for solving these problems in the multi-writer/reader shared memory model. However, in general, it is difficult to implement such algorithms in the single-writer/multi-reader shared memory model. This paper is a survey of the progress of algorithms for these problems in the asynchronous shared memory model.

1. Introduction

A distributed system is a collection of individual computing devices called processes or processors together with communication channels [6] [26]. Processes can communicate with other processes through communication channels in a network or through shared memory (variables) as a communication model. The communication model discussed in this paper is the latter type, i.e., the asynchronous shared memory model. The shared memory is an abstraction of asynchronous interprocess communication, where the senders and the receivers correspond to the writers and the readers, respectively. Each process in a distributed system is generally performed by its own program, but it is occasionally requested to collaborate with other processes.

Interactions between a process and its corresponding user are by input actions from the user to the process and by output actions from the process to the user. Each process is considered to be a state machine. All communication among the processes is via the shared memory (variables).

This model is known as an I/O automaton [2, 26]. In the multi-writer/reader shared memory model, the same shared variable may be read or written by different processes. In one step, a process can either read or write a single shared variable. Thus, the two actions involving process i and variable x are

(1) (read action) Process i reads variable x and the value read is used to modify the state of process i.

(2) (write action) Process i writes a value determined by the current state of process i to variable x.

The behavior of operation executions should be required to be consistent for all processes and interprocess communication. We therefore need a unified theory of shared memory consistency [13, 23, 24, 25, 31, 39]. Lamport [23] defined three categories, *safe*, *regular*, and *atomic*, for shared variables according to possible assumptions about what can happen in the concurrent case of read operations and write operations. A shared variable is *safe* if every read operation, that does not overlap with any write operation, returns the last value written to the shared variable. On the other hand, every read operation that overlaps with one or more write operations may return any value from the domain of the shared variable. A shared variable is said to be *regular* if every read operation returns either the last value written to the shared variable before the start of the read operation or a value written by one of the overlapping write operations. A shared variable is said to be *atomic* if it is regular with the additional property that read operations and write operations behave as if they occur in some total order. In this survey paper, we assume that all shared variables are *atomic*. That is, we assume that there is a possible linearization of the temporal order of read operations and write operations such that the linearization is consistent with the actual behavior of the system, although these operations may be physically overlapped.

Even if different processes try to write on the same variable at nearly the same time, one process's writing precedes the other process's writing. That is, the contents of the shared variable written by the earlier one is changed to the value written by the later one even if these two events occur very closely.

The mutual exclusion is one of the most fundamental problems for distributed computing, and historically it was first seriously studied by Dijkstra [8] as an important problem for a distributed operating system. It is the problem of how to allocate a single indivisible, non-sharable resource among n users. A user with access to the resource is modeled as being in a critical region (i.e., admitted state to use the resource). When a user is not involved in any way with the resource, it is said to be in the remainder region. In order to gain admittance to its critical region, a user executes a trying protocol. The duration from the state of executing the trying protocol to the entrance of the critical region is called the trying region. After the end of the use of the resource by a user, it executes an exit protocol. The duration of executing the exit protocol is called the exit region. Each user follows a cycle, moving from its remainder region to its trying region, then to its critical region, then to the exit region, and finally back to its remainder region. This cycle can be repeated.

The mutual exclusion problem is to design a fair and efficient algorithm to decide the temporal order among users wishing to use a shared resource [5, 6, 22, 23, 26, 29, 30]. The system to solve the mutual exclusion problem should satisfy the following conditions:

(1) There is no reachable system state in which more than one user is in the critical region.
(2) If at least one user is in the trying region and no user is in the critical region, then at some later time point some user enters the critical region.
(3) If a user is in the exit region, then at some later time point some user enters the remainder region.

Conditions (1), (2), and (3) above are called mutual exclusion, progress for the trying region, and progress for the exit region, respectively. If a mutual exclusion algorithm satisfies the following two conditions, too, it is said to be lockout-free:

(4) If all users always return the resource, then any user that reaches the trying region eventually enters the critical region.
(5) Any user that reaches the exit region eventually enters the remainder region.

An early algorithm for the mutual exclusion problem by Dijkstra [8] guarantees mutual exclusion and progress for both the trying region and the exit region, but it does not guarantees lockout-freedom. That is, the Dijkstra's algorithm may allow one user to be repeatedly granted for access to its critical region while other users trying to gain access never succeed in doing so.

A number of generalizations of the mutual exclusion problem have been proposed. These include the k-exclusion problem [3, 11, 27, 28], the group mutual exclusion problem [12, 19, 21, 35, 36], and the group k-exclusion problem [20, 33, 37, 38].

2. Mutual Exclusion in the Multi-Writer Shared Memory Model

The following procedure, *n-processME*, is well known lockout-free mutual exclusion algorithm, called the n-process algorithm in the multi-writer/reader shared memory model. The algorithm was first introduced by Peterson [29]. In order to analyze its time efficiency we describe it as a program in the I/O automata model as given in [26]:

```
procedure n-processME
shared variables
  for every k ∈ {1, . . . ,n − 1 }:
    turn (k) ∈ {1, . . . ,n }, initially arbitrary, writable and readable by all processes;
  for every i ∈ {1, . . . ,n }:
    flag (i) ∈ {0, . . . ,n − 1}, initially 0, writable by i and readable by all j ≠ i ;
process i:
  input actions {inputs to process i from user Ui }: try_i, exit_i ;
  output actions {outputs from process i to user Ui }: crt_i, rem_i ;
  ** Remainder region **
  try_i :
    for k = 1 to n − 1 do begin
      flag (i) := k;  turn (k) := i ;
      waitfor [∀j ≠ i : flag (j) < k ] or [turn (k) ≠ i ]  end ;
```

crit$_i$;
** Critical region **
exit$_i$:
flag (*i*):= 0;
rem$_i$;

In *n-processME*, *turn* variables (*turn* (*k*), $k = 1, \ldots, n-1$) are multi-writer variables, while *flag* variables (*flag* (*k*), $k = 1, \ldots n$) are single-writer shared variables. An exponential time bound, $2^{n-1}c + O(2^n nd)$ for the trying region of the *n*-process algorithm is given in [26], where n is the number of processes, d is an upper bound on the time between successive two steps, and c is an upper bound on the time that any user spends in the critical region. As shown in Theorem 2. 2, a finer analysis can give a polynomial time bound for the trying region of *n-processME*.

Theorem 2. 1 [26, 29]: *n-processME* solves the mutual exclusion problem, and it is lockout-free.

Theorem 2. 2 [16]: In *n-processME*, the time from when a particular process enters its trying region until it enters its critical region is at most $O(n^2c + n^4d)$.

The following two algorithms by Igarashi and Nishitani [16], *n-processFME1* and *n-process FME2*, are modifications of *n-processME*. By any of these modifications we can significantly speed up the original *n*-process algorithm.

procedure *n-processFME1*
shared variables: the same as the shared variables of *n-processME*
process *i* :
input/output actions: the same as the input/output actions of *n-processME*
** Remainder region **
try$_i$:
for $k = 1$ to $n - 1$ do begin
flag (*i*):= *k* ; *turn* (*k*):= *i* ;
waitfor [$\forall j \neq i$: *flag* (*j*) is not in {*k*, *k* + 1} or [*turn* (*k*) $\neq$ *i*]
end;
crit$_i$;
** Critical region **
exit$_i$:
flag (*i*):= 0;
rem$_i$;

procedure *n-processFME2*
shared variables: the same as the shared variables of *n-processME*
process *i* :
input/output actions: the same as the input/output actions of *n-processME*
** Remainder region **

```
try_i :
  for k = 1 to n - 1 do begin
    flag (i):= k ;  turn (k) := i ;
    waitfor [∀ j ≠ i : flag (j) < k] or [turn (k) ≠ i ]
  end;
crit_i ;
** Critical region **
exit_i :
  for k := n - 1 downto 1 do turn (k):= i ;
  flag (i):= 0;
rem_i ;
```

Theorem 2. 3 [16]: *n-processFME1* solves the mutual exclusion problem, and it is lockout-free.

Theorem 2. 4 [16]: In *n-processFME1*, the time from when a particular process has just entered its trying region until it enters its critical region is at most $(2n - 3)c + O(n^3d)$.

Theorem 2. 5 [16]: *n-processFME2* solves the mutual exclusion problem and it is lockout-free.

Theorem 2. 6 [16]: In *n-processFME2*, the time from when a particular process has just entered its trying region until it enters its critical region is at most $(n - 1)c + O(n^3d)$.

The k-exclusion problem is a natural generalization of the mutual exclusion. In k-exclusion, up to k processes are allowed to be concurrently inside the critical region, where corresponding users can use the resource. This generalization was defined by Fischer et al. in the shared memory model [11]. We can easily design algorithms for the k-exclusion problem if we are allowed to use multi-writer/reader shared variables as shown in [27, 28].

3. Tournament Algorithms in the Multi-Writer Shared Memory Model

The tournament algorithm for the mutual exclusion problem was originated by Peterson and Fischer [30]. Since the original one was not clearly described in the shared memory model, we quote here the algorithm from [26]. For simplicity, we assume that the number of processes, n is a power of 2. The algorithms in this chapter are described on the complete binary tree, called the tournament tree, with n leaves. The leaves of the complete binary tree are labeled as 0, 1, ... , $n - 1$ in binary representation. For each $0 \leqq i \leqq n - 1$, process i is associated with the leaf with label i. Each internal node in depth k of the complete binary tree, $0 \leqq k \leqq \log n - 1$, is labeled as the high-order k bits of the binary representation of any of its descendants. Note that the root of the tournament tree is labeled as λ, the null length string.

The following notations will be used to describe the original tournament algorithm and the modified tournament algorithm on the n-leaves tournament tree.

· *comp*(i, k) is the ancestor of i in depth k (i.e., the high order k bits of the binary

representation of i). Note that *comp*(i, *log* n) is the leaf associated with i.

- *role*(i, k) is the (k + 1)st high-order bit of i, (i.e., *role*(i, k) indicates whether the leaf i is a descendant of the left or right child of the node for comp(i, k)).
- opponents(i, k) is the opponents of process i in the depth k competition of process i (i.e., the set of processes with the high-order k bits as i and the opposite (k + 1)st bit.)
- *opposite*(i, k) is the son of *comp*(i, k) that is not an ancestor of i.

```
procedure n-tournamentME
shared variables
  for every binary string x of length at most log n - 1:
    turn(x) ∈ {0, 1}, initially arbitrary, writable and readable by those processes i
      for which x is a prefix of the binary representation of i ;
  for every i , 0 ≦ i ≦ n - 1:
    flag(i) ∈{0, 1, . . . , log n }, initially log n, writable by i and readable by all j ≠ i ;
process i :
  input actions {inputs to process i from user U_i } : try_i , exit_i ;
  output actions {outputs from process i to user U_i }: crit_i , rem_i ;
  ** Remainder region **
  try_i :
    for k = log n - 1 downto 0 do begin
      flag(i) := k ;
      turn(comp(i, k)) := role(i, k);
      waitfor [∀j∈opponents(i, k) : flag(j) > k ] or [turn(comp(i, k)) ≠ role(i, k)]
    end;
  crit_i ;
  ** Critical region **
  exit_i :
    flag(i) := log n ;
  rem_i ;
```

Theorem 3. 1 [26] : In *n-tournamentME*, the time from when a particular process enters its trying region until it enters its critical region is at most (n - *1*)c + O(n^2d).

Igarashi et al. [15] modified *n-tournamentME* to speed up the running of the original algorithm. For the modified n-tournament algorithm, **procedure** *n-tournamentFME*, each node of the n-leaves tournament tree, not only each leaf but also each internal node, is associated with a *flag* variable.

```
procedure n-tournamentFME
shared variables
  for every binary string x of length at most log n - 1:
    turn(x): the same as turn(x) in n-tournamentME ;
  for every binary string x of length at most log n:
```

flag(x) $\in$ {0, 1}, initially 0, writable by those processes i for which x is a prefix of the binary representation of i, and readable by those processes for which i is a descendant of the parent of x, but the i's bit, at the position corresponding to the last bit of x, is opposite from x;

process i:

input/output actions: the same as the input/output actions of *n-tournamentME*

** Remainder region **

try_i:

for k = log n - 1 downto 0 do begin

flag(*comp*(i, k + 1)) := 1;

turn(*comp*(i, k)) := *role*(i, k);

waitfor [*flag*(*opposite*(i, k)) = 0] or [*turn*(*comp*(i, k)) $\neq$ *role*(i, k)]

end;

$crit_i$;

** Critical region **

$exit_i$:

for k = 0 to log n - 1 do

flag(*comp*(i, k + 1)) := 0;

rem_i;

Theorem 3. 2 [15] : In *n-tournament FME*, the time from when a particular process enters its trying region until it enters its critical region is at most $(n - 1)c + O(nd)$.

4. Mutual Exclusion in the Single-Writer Shared Memory Model

In the single-writer/multi-reader shared memory model, each variable can be written by only one process, but may be read by any process. Algorithms described in the previous chapters use multi-writer shared variables. Because multi-writer shared variables are often difficult to implement, it is worth to study algorithms that use only single-writer shared variables. The first algorithm in this chapter was invented by Burns [7], and called Burns' algorithm (*BurnsME*).

procedure *BurnsME*

shared variables:

for every i, $1 \leq i \leq n$:

flag(i) $\in$ {0, 1}, initially 0, writable by i and readable by all $j \neq i$;

process i:

input/output actions: the same as *n-processME*

** Remainder region **

try_i:

L: *flag*(i) := 0;

for j, $1 \leq j \leq i - 1$ do

if *flag*(j) = 1 then goto L;

```
        flag(i) := 1;
        for j, 1 ≦ j ≦ i - 1 do
          if flag(j) = 1 then goto L;
M:      for j, i + 1 ≦ j ≦ n do
          if flag(j) = 1 then goto M;
  crit_i ;
  ** critical region **
  exit_i :
    flag(i) := 0;
  rem_i ;
```

Theorem 4. 1 [26] : *BurnsME* satisfies mutual exclusion, and guarantees the progress condition.

As stated in Theorem 4. 1, *BurnsME* solves the mutual exclusion in the single-writer/multi-reader shared memory model. However, it does not guarantee lockout-freedom. In the worst case, some process never obtains the admittance to enter its critical region even if it always wishes to do so. This is a serious deficiency of *BurnsME*.

The Bakery algorithm for the mutual exclusion problem guarantees lockout-freedom and a good time bound. It is accredited to Lamport [23], and works like a queue of customers in a bakery, where customers draw tickets. The following procedure n-Bakery is the Bakery algorithm quoted from [6, 26]. The relation among pairs of integers is used in the algorithm. The relation $(a, b) < (a', b')$ means that $a < a'$, or $a = a'$ and $b < b'$. The entry section of the Bakery algorithm begins with a part called the doorway, where processes in the trying region obtain their tickets. Then processes with their tickets proceed to execute the major part of the algorithm. If process i completes execution of the doorway, before process j begins the doorway, process i enters the critical region before j does so. In this sense, the Bakery algorithm satisfies the first-in first-served property.

```
procedure n-Bakery
shared variables
  for every i ∈ {1, . . . , n}:
    choosing(i) ∈ {0, 1}, initially 0, writable by i and readable by all j ≠ i ;
    ticket(i) ∈ N, initially 0, writable by i and readable by all j ≠ i ;
process i
  input actions {inputs to process i from user U_i } : try_i , exit_i ;
  output actions {outputs from process i to user U_i } : crit_i , rem_i ;
  ** Remainder region **
  try_i :
    choosing(i) := 1;
    ticket(i) := 1 + max_{j≠i} {ticket_j} ;
    choosing(i) := 0;
    for each j ≠ i do begin
      waitfor choosing(j) = 0;
```

waitfor *ticket(j)* = 0 or (*ticket(i)*, *i*) < (*ticket(j)*, *j*) end;
crit$_i$;
** Critical region **
exit$_i$:
ticket(i) := 0;
rem$_i$;

Theorem 4. 2 [23]: The Bakery algorithm satisfies mutual exclusion, and guarantees lockout-freedom.

Theorem 4. 3 [26]: The running time for the trying region by *n-Bakery* is bounded by $(n - 1)c + O(n^2d)$

An unattractive property of the Bakery algorithm is that it uses unbounded size of shared variables.

5. Mutual Exclusion Algorithms Based on Bounded Tickets

Takamura and Igarashi [32, 34] proposed a simple algorithm based on bounded tickets for the mutual exclusion problem in the asynchronous single-writer/multi-reader shared memory model. They initially modify the Bakery algorithm so that it requires only bounded size single-writer/ multi-reader shared variables. This provisional version guarantees mutual exclusion under the condition that at least one user, who is trying to use the resource or using the resource, can be observed at any point in time after a user first tries to use the resource. In order to remove this condition they use an additional process in their algorithm, *n-bmexc*. It guarantees lockout-freedom, mutual exclusion, and the first-in first-out property. The existence of an additional process may be unattractive feature, but the shared memory size for their algorithm is smaller than that of the algorithm given by Abraham [1] and that of the algorithm by Jayanti et al [18].

In the following procedure, a ticket held by a process is a pair of a ticket number and the process identifier. Function *scanticket* () scans all shared variables *ticket(j)* ($1 \leqq j \leqq n$) and returns the set of tickets observed by a process in its trying region. Function *rmax*(*S*) returns the largest ticket number in set *S* if *S* is not the empty, and otherwise it returns - 1. Function *prev*$_i$(*S*) returns the identifier of the largest process that is smaller than process *i* in the order of pairs of ticket numbers and process identifiers if process *i* is not the smallest one in *S*, and otherwise it returns an arbitrary identifier except for *i* itself.

procedure *n-bmexcl*
shared variables
for every $i \in \{1, 2, \ldots, n + 1\}$:
choosing(i) $\in \{0, 1\}$, initially 0, writable by *i* and readable by all $j \neq i$;
ticket(i) $\in \{-1, 0, \ldots, 4n - 2\}$, initially - 1 for $1 \leqq i \leqq n$ and 0 for $i = n + 1$, writable by *i* and readable by all $j \neq i$;

```
process i  (for 1 ≦ i ≦ n)
  input/output actions: the same as the input/output actions of n-Bakery
    try_i :
      choosing(i) := 1;
      ticket(i) := (1 + rmax(scanticket ())) mod 4n - 1;
      choosing(i) := 0;
      index := {1, 2, . . . , n + 1};
      while index ≠ φ do
      for each j ∈index do
        if choosing(j) = 0 then index := index - {j};
      j := prev_i (scanticket ());
      waitfor ticket(j) = - 1 or (ticket(i), i ) < (ticket (j), j );
  crit_i ;
  ** Critical region **
  exit_i :
    ticket (i) := - 1;
  rem_i ;
process n + 1
  input/output actions:  none
    repeat
      choosing (n + 1) := 1;
      ticket (n + 1) := (1 + rmax (scanticket () - |ticket (n + 1)|)) mod 4n - 1;
      choosing (n + 1) := 0;
      index := {1, 2, . . . , n, n + 1};
      while index ≠ φ do
        for each j ∈ index do
          if choosing (j) = 0 then index := index - {j};
      j := prev_{n+1} (scanticket());
      waitfor ticket (j) = - 1 or (ticket(n + 1), n + 1) < (ticket(j), j );
      waitfor |scanticket ()| ≧ 2;
    forever;
```

Theorem 5. 1 [32, 34] : For any execution by procedure *n-bmexcl*, mutual exclusion and lockout-freedom are satisfied, and the time from when any user requests to use the resource until it is allowed to be bounded by $(n - 1)c + O(nd)$. The total size of shared memory needed by it is $(n + 1)(3 + \log n)$ bits.

6. The Group Mutual Exclusion Problem

Group mutual exclusion is an interesting generalization of the mutual exclusion problem. This problem was introduced by Joung [20], and some algorithms for the problem have been proposed [12, 19, 20, 21]. Group mutual exclusion is required in a situation where a resource can be shared

by processes of the same group, but not by processes of different groups. As stated in [19], in some applications such as computer supported cooperative work (e.g., a video-conference system with an electronic white board), it is necessary to impose mutual exclusion on different groups of users in accessing a resource, while allowing users of the same group to share the resource. A combination of k-exclusion and group mutual exclusion was also studied [33, 38]. The algorithms given [19, 33, 37, 38] use multi-writer/reader shared variables that may be concurrently written in the asynchronous shared memory model. The algorithms given in [12, 21] use n^2 and n multi-writer/reader shared variables, respectively that are never concurrently written.

Group mutual exclusion can be described as the congenial talking philosophers' problem. We assume that there are n philosophers. They spend their time thinking alone. When a philosopher is tired of thinking, he/she attempts to attend a forum. We assume that there is only one meeting room. A philosopher wishing to attend a forum can do so if the meeting room is empty, or if some philosophers interested in the same forum as the philosopher in question are already in the meeting room. The congenial talking philosophers' problem is to design as an algorithm such that a philosopher wishing to attend a forum will eventually succeed in doing so. Philosophers interested in the same forum as the current forum in the meeting room should be encouraged to attend it. This type of performance is measured as concurrent frequency. It is undesirable that the maximum waiting time for a philosopher wishing to enter a forum is too long.

Takamura and Igarashi [35, 36] proposed two algorithms based on ticket orders for the group mutual exclusion problem in the asynchronous shared memory model. These algorithms are some modifications of the Bakery algorithm. They satisfy lockout freedom and a high degree of concurrency performance. Each of these algorithms uses single-writer shared variables together with only two multi-writer shared variables that are never concurrently written. One of their algorithms has another desirable property, called smooth admission. By this property, during the period that the resource is occupied by the leader called the chair, a process wishing to join the same group as the leader's group can be granted use of the resource in constant time. The full description of these algorithms is given in [32, 35, 36].

The concurrency performance of the algorithms by Takamura and Igarashi [35, 36] is also superior to the algorithms in [12, 21], since their algorithms have properties of unbounded concurrent frequency, overtaking admission, and a property of smooth admission. However, they are not *wait-free* in the exit region. A *wait-free* operation means that it finishes its execution within a bounded number of its own steps, irrespective of the presence of other operation execution and their relative speeds [25, 39]. An execution by each of their algorithms takes $c + O(nd)$ time in the exit region. This is obviously a disadvantage of them. Another disadvantage of their algorithms is that the ticket domain is unbounded, as in the Bakery algorithm. At present we do not know whether we can simply modify their algorithms by using a similar technique given in [1, 17, 30] so that the ticket domain is bounded.

7. Bounded Concurrent Timestamps

It is only required that every process keeps track of which process wrote last in order to

determine the temporal orders among objects written in shared variables. It is easy to realize this function in the multi-writer/reader shared memory model. Specifically, the *turn* function, used in the algorithms in Chapters 2 and 3, allows each process in the trying region to decide if it is the last one to enter there. However, the realization of multi-writer/reader shared variables requires significant overhead. If we are allowed to use unbounded timestamps, the actions of multi-writer/reader shared variables can be easily simulated by single-writer shared variables. Israeli and Li raised the question of the general and universal notion of a bounded concurrent timestamp system, and eventually solved it by tracking the order of events in a concurrent system [17]. Israeli and Li constructed a bit-optimal bounded timestamp system for sequential operation executions. As stated in [39], their sequential timestamp system was published in [17], but the preliminary timestamp system given in the conference proceedings has not been published in journal form yet. The first generally accepted solution to the concurrent case of the bounded timestamp system is due to Dolev and Shavi [9]. Some other constructions of bounded concurrent time stamps can be found, for example, in [10, 14].

The bounded concurrent timestamp system is a useful tool when we design algorithms for determining the temporal order of objects written in single-writer shared variables. For example, Afek et al. [2, 3] gave a first-in and first-enable solution to the k-exclusion problem using bounded concurrent time-stamps. However, bounded concurrent time-stamp schemes usually contain complicated constructions. In general, a formal proof of the correctness of such a construction is very hard to be clearly described. In fact, early versions of some conference papers about bounded concurrent timestamps did not promptly appear as journal papers due to the doubt of their correctness.

The multi-writer/reader shared variables, called *turns*, used in [15, 16, 27 - 30] serve to decide the temporal order among waiting processes. There, every process wishing to use a resource can write its identifier to an appropriate multi-writer/reader variable, and by checking the value of this variable any process can easily decide which process is the last writer, or implicitly, which process was the last to enter the trying region. The *turn* variable/function is useful to solve problems such as the mutual exclusion problem and the k-exclusion problem, in the asynchronous shared memory model where a determination of a temporal order of events is required. Altman et al. [4] proposed an algorithm, called procedure *n-turn*, for the *turn* function in the single-writer shared memory model. The algorithm does not use time-stamps, and it satisfies lockout avoidance, even in the presence of up to $n - 2$ process failures of stopping type. By using procedure *n-turn*, the algorithms given in [15, 27 - 30] can also be implemented, without using timestamps, in the single-writer/multi-reader shared memory model. However, the formal proof of its correctness seems to be complicated. The informal proof given in [4] should be improved although we are convinced the procedure *n-turn* is a correct algorithm.

8. Concluding Remarks

We have described a number of previous works on algorithms for determining the temporal orders among objects written in shared variables. Each construction of these algorithms contains interesting ideas, but they are not optimal. We may be requested to design more

efficient algorithms concerning temporal relations in the asynchronous distributed systems. It is particularly interesting to find efficient and clear techniques for simulating multi-writer shared variables by single-writer shared variables. These would be worthy of further investigation.

References

[1] U. Abraham, "Bakery algorithms", Technical Report, Department of Mathematics, Ben Gurion University, Beer-Sheva, Israel, 2001.

[2] Y. Afek, H. Attiya, D. Dolev, E. Gafni, M. Merritt, and N. Shavit, "Atomic snapshots of shared memory", *J. of the ACM*, vol.40, pp.873-890, 1993.

[3] Y. Afec, D. Dolev, E. Gafni, M. Merritt, and N. Shavit, "A bounded first-in, first-enable solution to the L-exclusion problem", *ACM Transactions on Programming Languages and Systems*, vol.16, pp.939 - 953, 1994.

[4] T. Altman, Y. Igarashi, and M. Omori, "A turn function scheme realized in the asynchronous single-writer/multi-reader shared momory model, *14th International Symposium on Algorithms and Computation (ISAAC 2003)*, Kyoto, Japan, *LNCS* 2906, pp.454-463, 2003.

[5] J. H. Anderson, "Lamport on mutual exclusion: 27 years of planting seeds", *Proc. 27th Annual ACM Symposium on Principles of Distributed Computing*, pp.3-12, Newport, Rhode Island, 2001.

[6] H. Attiya and J. Welch, "Distributed Computing: Fundamentals, Simulations and Advanced Topics", McGraw-Hill, New York, 1998.

[7] J. E. Burns, "Mutual exclusion with linear waiting using binary shared variables", *ACM SIGACT News*, vol.10, pp.42-47, 1978.

[8] E. W. Dijkstra, "Solution of a problem in concurrent programming control", *Commun. of the ACM*, vol.8, p.569, 1965.

[9] D. Dolev and N. Shavit, " Bounded concurrent time-stamp systems are constructable", Siam J. Comput., vol.26, pp.418-455, 1997 (Its preliminary version is in Proc. *21st Annual ACM Symposium on the Theory of Computing*, New York, pp.454-465, 1989.

[10] C. Dwork and O. Waarts, "Simple and efficient bounded concurrent time stamping or concurrent time stamp systems are comprehensible, *24th Annual ACM Symposium on the Theory of Computing*, Victoria, B.C., pp.655-666, 1992.

[11] M. J. Fischer, N. A. Lynch, J. E. Burns, and A. Borodin, "Resource allocation with immunity to limited process failure", *20th Annual Symposium on Foundations of Computer Science*, San Juan, Puerto Rico, pp.234-254, 1979.

[12] V. Hadzilacos, "A note on group mutual exclusion", *Proc. 12th Annual ACM Symposium on Principles of Distributed Computing*, pp.100-106, Newport, Rhode Island, 2001.

[13] S. Haldar and K. Vidyasankar, "On Specification of Read/Write Shared Variables", *J. of the ACM*, vol.54, pp.31-1 - 31-19, 2007.

[14] S. Haldar and P. Vitanyi, "Bounded concurrent time stamps using vector clocks", *J. of the ACM*, vol.49, pp.101-126, 2002.

[15] Y. Igarashi, H. Kurumazaki, and Y. Nishitani, "Some modifications of the tournament algorithm for the mutual exclusion problem", *IEICE Trans. Inf. and Syst.*, vol.E82-D, pp.368-375, 1999.

[16] Y. Igarashi and Y. Nishitani, "Speedup of the *n*-process mutual exclusion algorithm", *Parallel Processing Letters*, vol.9, pp.475-485, 1999.

[17] A. Israeli and M. Li, "Bounded time-stamps", *Distributed Computing, vol.6, pp.205-209, 1993 (Its preliminary version is in Proc. 28th IEEE Symp. Foundations of Computer Sceience*, Los Angeles, pp.371-382, 1987).

[18] P. Jayanti, K. Tan, G. Friedland, and A. Katz, "Bounded Lamport's bakery algorithm", *Proc. SOFSEM' 2001*, LNCS, vol.2234, pp.261-270, 2001.

[19] Y. J. Joung, "Asynchronous group mutual exclusion", *Distributioon Computing*, vol.13, pp.189-206, 2000.

[20] Y. J. Joung, "The congenial talking philosophers problem in computer networks", *Distributed Computing*, vol.15, pp.155-175, 2002.

[21] P. Keane and M. Moir, "A simple local-spin group mutual exclusion algorithm", *IEEE Trans. Parallel Distrib. Syst.*, vol.12, pp.673-685, 2001.

[22] L. Lamport, "A new solution of Dijkstra's concurrent programming problem", *Comm. of the ACM*, vol.17, pp.453-455, 1974.

[23] L. Lamport, "The mutual exclusion problem, Part II", *J. of the ACM*, vol.33, pp.327-348, 1986.

[24] L. Lamport, "On interprocess communication, Part II: Algorithms", *Distributed Computing*, vol.1, pp.86-101, 1986.

[25] K. Li, I. Tromp and P. M. B. Vitanyi, "How to share concurrent wait-free variables", *J. of the ACM*, vol.43, pp.723-746, 1996.

[26] N. A. Lynch, "Distributed Algorithms", Morgan Kaufmann, San Francisco, California, 1996.

[27] K. Obokata, M. Omori, K. Motegi, and Y. Igarashi, "A lockout avoidance algorithm without using time-stamps for the *k*-exclusion problem, *7th Annual International Computing and Combinatorics Conference (COCOON 2001)*, Guilin, China, LNCS, vol.2108, pp.571-575, 2001.

[28] M. Omori, K. Obokata, K. Motegi, and Y. Igarashi, "Analysis of some lockout avoidance algorithms for the *k*-exclusion problem", *Interdisciplinary Science*, vol.2, pp.187-198, 2002.

[29] G. L. Peterson, "Myths about the mutual exclusion problem", *Information Processing Letters*, vol.12, pp.115-116, 1981.

[30] G. L. Peterson and M. J. Fischer, "Economical solutions for the critical section problem in a distributed system", *9th Annual ACM Symposium on Theory of Computing*, Boulder, Colorado, pp.91-97, 1977.

[31] R. C. Steinke and G. J. Nutt, "A unified theory of shared memory consistency", *J. of the ACM*, vol. 51, pp.800-848, 2004.

[32] M. Takamura, "Algorithms for the Mutual Exclusion Problem and Its Generalization", Ph.D dissertation, Gunma University, Kiryu, Japan, 2004.

[33] M. Takamura, T. Altman, and Y. Igarashi, "Speedup of Vidyasankar's algorithm for the group *k*-exclusion problem", *Information Processing Letters*, vol.91, pp.85-91, 2004.

[34] M. Takamura and Y. Igarashi, "Simple mutual exclusion algorithms based on bounded tickets on the asynchronous shared memory model", *8th Annual International Computing and Combinatorics Conference (COCOON 2002)*, Singapore, *LNCS*, vol.2387, pp.259-268, 2002.

[35] M. Takamura and Y. Igarashi, "Group mutual exclusion algorithms based on tickets order" *9th International Computing and Combinatorics Conference (COCOON 2003)*, Big Sky, Montana, *LNCS*, vol.2697, pp.232-241, 2003.

[36] M. Takamura and Y. Igarashi, "Highly concurrent group mutual exclusion algorithms based on ticket orders", *IEICE Transactions on Information and Systems*, vol.E87-D, pp.322-329, 2004.

[37] K. Vidyasankar, "A highly concurrent group mutual *l*-exclusion algorithm", *Proc. 12th Annual ACM Symposium on Principles of Distributed Computing*, p.130, Monterrey, CA, 2002.

[38] K. Vidyasankar, "A simple group mutual *l*-exclusion algorithm", *Information Processing Letters*, vol.85, pp.79-85, 2003.

[39] P. Vitanyi, "Registers", Research Memo, University of Amsterdam, 2006.