Bakery Algorithms

Uri Abraham

Department of Mathematics

Ben Gurion University

Beer-Sheva, Israel

abraham@indigo.bgu.ac.il

May 18, 2001

Abstract

An approach to proving higher level properties of distributed protocols is suggested here in which a proof consists of two stages: In the higher-level stage, abstract properties of system executions are assumed and their desired consequences are proved. At the lower-level stage these abstract properties are shown to hold in every execution of the protocol.

The paper studies the Bakery Algorithm of Lamport and some variants (these are all mutual exclusion protocols), and its main contribution is a protocol which is similar to the Bakery Algorithm but only uses bounded values. Another variant uses local clocks to coordinate the protocol's execution, and conditions on the clocks are determined that suffice for the protocol's correctness.

In what sense is our bounded algorithm similar to the unbounded Bakery Algorithm? In that both satisfy the "Bakery Definability property" (which is a strong form of the First Come First Serve property).

An advantage of this two stage approach is demonstrated by the correctness proofs given to these variants of the Bakery Algorithm: Since they are shown to share with the Bakery Algorithm the same higher-level properties, their correctness proofs are reduced to only one stage (the lower level).

Categories and Subjects Descriptors: D.2.4 [**Software Engineering**]: Programs Verification—*correctness proofs*; F.3.1 [**Logic and Meanings of Programs**]: Specifying and Verifying and Reasoning about Programs.

1 Preface

The Critical Section Problem is so well known that it suffices to refer to any operating-system text-book for its description. One solution, the Bakery Algorithm (Lamport [9], and a variant in [10]), stands up for its naturalness and simplicity. (Perhaps I should say deceiving simplicity since concurrent protocols can be very tricky sometimes.) In this solution (reproduced below in Section 2) unbounded time-stamps (natural numbers) are used to coordinate the processes' access to their critical-sections.

This paper investigates the Bakery Algorithm of Lamport [9], and introduces a new protocol which is similar to the Bakery Algorithm, but only uses bounded values. The similarities are expressed in terms of higher-level properties of system executions which the two protocols share. One such property is defined and called the "Bakery Definability Property".

The mutual exclusion and the deadlock freedom are the main properties that any solution of the critical section problem is required to satisfy, but other important properties have been investigated as well, such as the First-Come, First-Served (FCFS) property (in Lamport [12] part II, for example). We will introduce here another property, the Bakery Definability Property, which implies both the mutual exclusion and the FCFS properties. Intuitively, it says that the entry order into the critical sections is determined by the doorways alone. That is, if in two runs of the Bakery Algorithm the doorways are the same, then the (linear) order of the critical section events in these two runs is also the same.

The correctness proof is separated into two stages. At the lower-level, the text of the algorithm is used to prove some higher-level properties (or specifications), and at the higher level stage it is shown how these specifications imply the Bakery Definability property (and hence the mutual exclusion and the FCFS properties). Since the higher-level properties are shared by all the protocols presented in this paper, the correctness proofs need not repeat this common part.

The paper begins with an exposition of the Bakery Algorithm (and a variant) and it then reviews the notions "system execution" and higher and lower "views" (Lamport [11]). The Introduction also contains definitions of terms (such as FCFS) that are used in this article. The Bakery Definability property is defined in subsection 2.2. In Section 3, the higher level specifications are defined and proved to hold for the Bakery Algorithm. In Section 4, these are used to derive the mutual exclusion and the FCFS properties. In Section 5, bounded values replace the unbounded natural numbers used in the Bakery Algorithm.

The motivation for this paper came from Afek et al. [3], in which the problem of l-exclusion is addressed and solved with only bounded size registers. In their solution, bounded timestamps are used (as defined by Israeli and Li [8], and constructed by Dolev and Shavit [6]). For $l = 1$, the l-exclusion is just the mutual exclusion, and their solution satisfies in fact the Bakery Property. Our solution here is more economical than the $l = 1$ case of the Afek et al. paper, if the price of implementing those bounded timestamps is also taken into account.

2 Introduction

This section introduces the main notions that will be needed in the proof, and two protocols (the Bakery Protocol and the Local Clock Protocol) will provide concrete examples to these notions.

Assume N processes, $P_1, \ldots, P_N$ that may need from time to time to access a 'critical section' in the code they execute. The role of a mutual exclusion protocol is to make these accesses disjoint in time. Each P_i may execute the protocol any (finite or infinite) number of times. Its other activities can either be omitted or lumped together as noncritical-section, "external", events (which can be non-terminating).

We make a distinction between the Mutual Exclusion Algorithm (Fig-

Mutual Exclusion Algorithm

l_1 forall $i \in \{1, \ldots, N\}$ concurrently do
$l_{1.1}$ non-critical section
$l_{1.2}$ Protocol for P_i

Figure 1: The Bakery Algorithm for N processes.

ure 1) which is the general framework of the processes that cycle through non-critical sections and controled access to their critical sections, and the protocol (such as the Bakery Protocol which is the protocol that each proces excutes instead of directly accessing its critical section.

A rather simplistic protocol is one in which each process P_i can access a clock and in one atomic step both read it and write the time on register R_i.

Simplistic Protocol

l_1 write on R_i the time value of *clock*;
l_2 forall $j \neq i$ repeat read(R_j, n)until $n > R_j$;
l_3 Critical Section;
l_4 write ∞ on R_i.

The atomicity requirement made here is unacceptable, of course, because reading the clock and writing the value on R_i are two separate operations. If we substitute line l_1 with
$l_{1.1}$ $n = clock$;
$l_{1.2}$ write n on R_i;
then the mutual-exclusion property may be violated since operations of another process may be interleaved between line 1.1 and 1.2. (It is an interesting exercise to see exactly how this can happen.) To correct this simplistic protocol a boolean register G_i is added to process P_i and the following Local Clock Protocol is obtained (based on the Bakery Protocol of Lamport described below).

The *local clocks protocol* assumes that each process can call a procedure called *clock*, which returns a natural number representing the time passed since the system begun executing. So line

$$n := clock$$

assigns this number to n For definiteness, we have assumed that *clock* returns integers. The clocks used by the processes are assumed to be local and we to ensure the correct functioning of the protocol we assume that if r_1 and r_2 are invocation of this *clock* procedure (possibly by different processes) and if r_1 completely precedes r_2, then the value returned by r_1 is strictly smaller than that of r_2. (If r_1 andd r_2 overlap, even partially, then no assumptionss are made on the possible values returned.)

Each process P_i uses registers G_i and R_i for which it is the only writer and which other processes may concurrently read. These registers are accessed only by the protocol statements (and not by any external statements that the processes may execute). G_i is boolean, initially set to 0. The domain of R_i is the set of natural numbers and an additional value ∞, which is its initial value. (It is convenient to assume, for each register, an *Initialization* event that writes the initial value). The registers are required to be safe (which means that a read overlapping a write may return any value, see below for a full definition). The forall command in the protocol may be executed concurrently for the indices $j \in \{1 \dots N\}$. The variable n is local to each process, it is an array $n[1], \dots, n[N]$ of natural numbers or ∞.

The Protocol for P_i (The Local Clocks protocol)

l_1 write$(G_i, 1)$

l_2 $n := clock$

l_3 write(R_i, n)

l_4 write$(G_i, 0)$

l_5 forall $j \neq i$

1. repeat read(G_j, g) until $g = 0$
2. repeat read(R_j, k) until $(k, j) > (n, i)$

l_6 Critical Section

l_7 write(R_i, ∞).

Let's go over the protocol line by line to clarify the semantics of the instructions. The first instruction is to write value 1 onto register G_i. I prefer this style to $G_i := 1$ which treats register G_i though it were a local variable. The second line invokes procedure *clock* and assigns its value to the local variable n. Line l_3 and l_4 are writes on registers R_i and G_i. These registers are assume to be safe (a notion which will be explained later). Line l_5 consists of executing in any order (and in fact even concurrently) the two lines that follow in which index j varies over all indices of the process except for i itself (the executing process). The repeat instruction is a loop that executes at least once, and possibly an infinite number of times.

The instruction read(R_j, k), for example, assigns to variable k the value returned in reading register R_j.

Condition $(k, j) > (n, i)$ that stops the second loop means that $k > n$, or else $k = n$ and $j > i$.

Now we describe the Bakery Protocol of Lamport. The Bakery Protocol needs no clocks at all. To execute the Bakery Protocol (see below), each process P_i has two registers R_i and G_i as in the Local Clocks Protocol.
The Protocol for P_i (The Bakery Protocol)

l_1 write$(G_i, 1)$

$l_{2.1}$ forall $j \neq i$ read$(R_j, n[j])$

$l_{2.2}$ let $n[i]$ be any integer bigger than
$\max\{\, n[j] \;:\; 1 \leq j \leq N, j \neq i \text{ and } n[j] \neq \infty\}$

l_3 $\mathsf{write}(R_i, n[i])$

l_4 $\mathsf{write}(G_i, 0)$

l_5 $\mathsf{forall}\ j \neq i$

1. $\mathsf{repeat\ read}(G_j, g)\ \mathsf{until}\ g = 0$
2. $\mathsf{repeat\ read}(R_j, k)\ \mathsf{until}\ (k, j) > (n[i], i))$

l_6 Critical Section

l_7 $\mathsf{write}(R_i, \infty)$.

The order relation used in line $l_5(2)$ is defined by $(n, j) > (m, i)$ iff $n > m$, or $n = \infty$, or $n = m$ and $j > i$.

The code of many mutual exclusion protocols is divided into four consecutive parts (see Lamport [12]): the doorway, the wait, the critical section, and the exit part. In the Bakery Protocol and the Local Clock protocol, lines $l_1 - l_4$ constitute the doorway, line l_5 is the wait part, line l_6 is the critical section, and line l_7 the exit. The doorway contains no wait statements. There is no bound on the execution time of the wait part, and there cannot be such a bound since the other processes are allowed to stay as long as they wish in their critical sections—provided that this stay is of finite duration. It is natural to require that the exit part is bounded. Formally, the demand is that there is a fixed bound, k, such that any execution of the doorway or the exit parts never takes more than k many read/write steps

I find it convenient to add a "clock" part to this division, which is typical to the family of protocols investigated here. The clock part is obtained by omiting lines l_1 and l_4 from the doorway: it is line $l_2 - l_3$ in the Bakery Protocol and the Local Clock protocol. In this part, each process determines its timestamp and shows it to the other processes.

It is intuitively clear that these two protocols, the Local Clocks and the Bakery protocols, are very similar. Yet there is a marked structural difference. If $x <_{\mathcal{S}} y$ are two *door* events in the Local Clock protocol, then the time obtained in the *clock* invocation in y is a greater number than the time obtained in x; simply because y occurs later and we make this assumption on the local clocks that later invocations return greater numbers. For the Bakery Protocol this is not necessarily the case. It can well happen that $x <_{\mathcal{S}} y$ are *door* events and yet the timestamp number obtained in line $l_{2.2}$ in y is not greater than that of x. The reason for this is that in reading register R_i of x, y obtaiend not the value written by x but a later value. A major question that we try to answer here is this. Find a common framework, a unified proof, in which the two protocols can be proved to be correct. Higher level properties that hold in system executions of these protocols will provide such a framework as we shall see.

The following subsection contains a short introduction to system executions, which are used here to model "runs" of protocols.

2.1 System executions

If some omnipresent agent records the activity of the processes in one run, the result is abstracted into a mathematical structure of a particular type, called *system execution*. Any mathematical structure consists of a universe, relations, and functions on that universe. System executions are two-sorted structures, containing "events" and "atemporal" objects. An example of an event is a read or write operation execution, and an example of an atemporal object is the value (a number) of an event. Atemporal objects are so called because they are not located in time. A *system execution* is a structure $\mathcal{S} = (E_{\mathcal{S}}, X_{\mathcal{S}}; <_{\mathcal{S}}, F_{\mathcal{S}})$ consisting of the following elements: $E_{\mathcal{S}}$ is a set of events, $<_{\mathcal{S}}$ is the temporal precedence relation on $E_{\mathcal{S}}$, $X_{\mathcal{S}}$ is the set of atemporal objects, and $F_{\mathcal{S}}$ is a collection of predicates and functions on $E_{\mathcal{S}} \cup X_{\mathcal{S}}$ which are used to describe the properties of the execution. An example of such a predicate is one that tells about an event whether it is a 'write' event or not. An example of a function is $Value(x)$: it is a function into the integers (assumed to be in $X_{\mathcal{S}}$) that gives for a write or read event its value.

The temporal precedence relation $x <_{\mathcal{S}} y$ indicates that event x ends before y begins. We write $x \leq_{\mathcal{S}} y$ as a shorthand for

$$x <_{\mathcal{S}} y \text{ or } x = y.$$

As the events are not assumed to be linearly ordered in time, $<_{\mathcal{S}}$ is a partial ordering of $E_{\mathcal{S}}$, and it is required to satisfy the following two properties:

1. $<_{\mathcal{S}}$ is transitive and irreflexive on $E_{\mathcal{S}}$.

2. If $a <_{\mathcal{S}} b$, $b \not>_{\mathcal{S}} c$, and $c <_{\mathcal{S}} d$, then $a <_{\mathcal{S}} d$. Here $b \not>_{\mathcal{S}} c$ is the negation of $c <_{\mathcal{S}} b$. (This property was investigated by Russell and Wiener, and we will call it the Russell-Wiener property; see Wiener [14]).

Any relation satisfying these two properties is called an interval ordering. These orderings are characterized as those obtained by taking a collection of intervals of a linear ordering and defining interval A to precede B if all the points of A are smaller than all the points of B. (For a complete treatment, see the book by Fishburn [7], and also Abraham [2]). It is easy to see that interval orderings must also satisfy the following:

$$a <_{\mathcal{S}} b, b \not>_{\mathcal{S}} c \rightarrow a \not>_{\mathcal{S}} c,$$

$$a \not>_{\mathcal{S}} b, b <_{\mathcal{S}} c \rightarrow a \not>_{\mathcal{S}} c.$$

The events in $E_{\mathcal{S}}$ are of two types: terminating and nonterminating. Intuitively, a terminating event has a finite duration while the temporal interval of a nonterminating event is unbounded. Formally, there is in $F_{\mathcal{S}}$ a predicate that tells whether or not an event is terminating. If $x <_{\mathcal{S}} y$ then x is required to be terminating. Each event in $E_{\mathcal{S}}$ has only finitely many predecessors, but, for terminating events, the following stronger *Finiteness Property* (formulated by Lamport) is required to hold: For every terminating $e \in E_{\mathcal{S}}$

$$\{ x \in E_{\mathcal{S}} \ : \ x \not>_{\mathcal{S}} e \} \text{ is finite.}$$

We assume that all the events that correspond to instructions of the protocol, and in particular the critical section events, are terminating. The higher-level events, which will be considered in the following subsection, may be nonterminating. In fact, a higher level event is a collection of lower-level events, and it will be nonterminating if it is infinite. Events that are outside of the protocols (sometimes called non-critical events) do not interest us and do not appear in our system executions—but this is a mater of taste and the reader who wants so may introduce such events which may be nonterminating.

The registers used by the Bakery and the local clocks protocols (namely the R_i's and G_i's) are assumed to be *safe* (Lamport [11] and see the definition below). Let R be a register, and assume that the set of writes on R in $\mathcal{S}$ is linearly ordered (by $<_{\mathcal{S}}$), and also that there exists an initial write onto R that precedes all reads. The "safeness" of R is defined as follows.

Definition 2.1 *Register R is "safe" in system execution $\mathcal{S}$ if the following hold: For every read r either r overlap with some write w (i.e., $r \not>_{\mathcal{S}} w$ and $w \not>_{\mathcal{S}} r$), or else the value returned by r equals the value written by the latest write preceding r. That is, $Value(r) = Value(w)$, where w is the $<_{\mathcal{S}}$-rightmost write on R such that $w <_{\mathcal{S}} r$. (Without assumption of an initial write we would have to specify that reads that precede all writes must return the initial value).*

Technically, it is convenient to express the safeness of a register by assuming that $F_{\mathcal{S}}$ contains a "return" function, ω, that associates with each read r of R a write $w = \omega(r)$ on R such that:

1. $\omega(r) \not>_{\mathcal{S}} r$.
2. There is no write w_1 onto R such that $\omega(r) <_{\mathcal{S}} w_1 <_{\mathcal{S}} r$.
3. If $\omega(r) <_{\mathcal{S}} r$, then $Value(r) = Value(\omega(r))$.

Since the writes on R are linearly ordered, the existence of such a function ω is equivalent to the safeness of R. A function satisfying properties 1 and 2 above is called "regular".

In the second part of this paper we shall deal with serial registers. A register R is *serial* in a system execution iff it is safe and the read/write events are linearly ordered.

2.2 Lower level views

Any particular execution of the Bakery Algorithm is described by a system execution $\mathcal{S}$ in which the lower level events that correspond to the protocol's instructions are recorded. Thus the set of events E contains read/write events, critical section, and non-critical section events. Several predicates and functions are needed in $\mathcal{S}$. For example $P_i(x)$ says that event x is executed by process P_i. We view process P_i as the set of all events that belong to predicate P_i. So we do not distinguish the predicate from its extension and we can also write $x \in P_i$ instead of $P_i(x)$. Another predicate is $cs(x)$ which says that event x corresponds to an execution of line l_6 of the Bakery Protocol. The functions *Value* and ω are used to state that the registers are safe. The system executions that interest us are only those in which the events by each P_i correspond to repeated executions of the Bakery Protocol.

2.3 Higher level views

In fact, to every run of the system two types of events are associated: The lower-level and the higher-level events. The lower-level events, as described above, contain the read/write accesses to the registers, and the critical sections. A higher-level event corresponds to a *collection* of lower-level events. Below we will be specific as to which higher-level events are taken, but any set of events in $E_{\mathcal{S}}$ can be grouped into a single entity called a higher-level event. A collection of higher-level events may belong to a system execution which is called a higher-level view of $\mathcal{S}$. Denoting this higher-level system execution by $\mathcal{H}$, any higher-level event $X \in E_{\mathcal{H}}$ is thus a subset of $E_{\mathcal{S}}$, and the precedence relation on these events is defined by

$$X <_{\mathcal{H}} Y \text{ iff } \forall x \in X \; \forall y \in Y \; (x <_{\mathcal{S}} y).$$

A higher-level event $X \in E_{\mathcal{H}}$ can be infinite and it is not required that distinct events in $E_{\mathcal{H}}$ are disjoint (subsets of $E_{\mathcal{S}}$). However, a lower-level $x \in E_{\mathcal{S}}$ may belong to only finitely many higher-level events. This ensures the finiteness property of $\mathcal{H}$. Indeed, a higher-level event $X \in E_{\mathcal{H}}$ is defined to be 'terminating' in $\mathcal{H}$ iff it is a *finite* collection of terminating lower-level events. The finiteness property of $\mathcal{H}$ is now a consequence of the pigeonhole principle. Also, if $X <_{\mathcal{H}} Y$, then X is finite, by the finiteness property in $\mathcal{S}$. (More details about the concept "higher level" can be found in Lamport [11] where it is defined.)

It is often useful to relate a lower-level event $a \in E_{\mathcal{S}}$ to a higher-level event $X \in E_{\mathcal{H}}$: For example $a <_{\mathcal{H}} X$ means that $\forall x \in X \; a <_{\mathcal{S}} x$. (similarly, $X <_{\mathcal{H}} a$ may be defined).

We will also use formulas of the form $begin(X) <_{\mathcal{H}} Y$ There is no need to introduce $begin(X)$ and $end(X)$ as events of $E_{\mathcal{H}}$ if we define

$$begin(X) <_{\mathcal{H}} Y \text{ iff for all } B \in E_{\mathcal{H}} \text{ such that } B <_{\mathcal{H}} X, \; B <_{\mathcal{H}} Y.$$

$$X <_{\mathcal{H}} end(Y) \text{ iff for all } B \in E_{\mathcal{H}} \text{ such that } Y <_{\mathcal{H}} B, \; X <_{\mathcal{H}} B.$$

The following simple lemma will be frequently used.

Lemma 2.2 *If $X <_{\mathcal{H}} Y \not>_{\mathcal{H}} Z$, then $X <_{\mathcal{H}} end(Z)$.*

The proof is in one line: For any event E, if $Z <_{\mathcal{H}} E$, then the Russell-Wiener property implies that $X <_{\mathcal{H}} E$. (We use here the easily established fact that the Russell-Wiener formula holds in $\mathcal{H}$ if it holds in $\mathcal{S}$.

In case $\mathcal{S}$ describes an execution of the Bakery Algorithm, the higher level system execution $\mathcal{H}$ of $\mathcal{S}$ is obtained by grouping the following subsets of $E_{\mathcal{S}}$.

1) A set $A \subseteq E_{\mathcal{S}}$ is said to form a *protocol execution* by P_i of the Bakery Protocol if it includes consecutive events in P_i that correspond to an execution of the algorithm (i.e., executions of lines l_1 to l_7 of the algorithm, or lines 1 to 5 if one of the two repeat/until loops is not terminating, and then A is nonterminating protocol execution). For every $A \subseteq E_{\mathcal{S}}$ that represents an execution of the Bakery Protocol, put $A \in E_{\mathcal{H}}$. At this stage of the proof we do not know yet that the repeat loops of line l_5 are always terminating and we allow for nonterminating loops as well. So, executions of the protocol may either be terminating or nonterminating. We assumed that all the lower-level events of the Bakery Protocol are terminating, and thus a higher-level event is nonterminating iff it is infinite.

2) For a protocol execution A as above, the doorway of A is the set of events corresponding to lines $l_1 - l_4$. More important for us is the higher-level event $clock(A) \in E_{\mathcal{H}}$ consisting of the events in A related to lines l_2, l_3. Also, $wait(A) \in E_{\mathcal{H}}$ is the subset of A that contains all the reads corresponding to the execution of line l_5. ($wait(A)$ is nonterminating iff it is infinite.) If A is terminating then $cs(A)$ is the higher level event encompassing the execution of the critical section in A, and $exit(A)$, is the singleton containing the write of ∞ onto R_i. The higher-level events $clock(A)$, $wait(A)$, $cs(A)$, and $exit(A)$ are added to the higher-level system execution $\mathcal{H}$, and the functions *clock*, *wait*, *cs* and *exit* are put in $F_{\mathcal{H}}$ (*clock* and *wait* are defined on protocol events, and *cs* and *exit* on terminating protocol events. Events of the form $clock(A)$, for example, are called *clock* events, and we assume in $F_{\mathcal{H}}$ a predicate that tells the type of each event).

3) The higher-level initializing events are the singletons $\{a\}$ where a is in $\mathcal{S}$ an initializing write onto one of the registers.

The system execution $\mathcal{H} = \langle E_{\mathcal{H}}, X_{\mathcal{H}}; <_{\mathcal{H}}, F_{\mathcal{H}} \rangle$ thus obtained is called a higher-level view of $\mathcal{S}$. The *mutual exclusion* in $\mathcal{H}$ in the following statement: *For any distinct* $A_1, A_2 \in E_{\mathcal{H}}$ *representing terminating executions of the protocol,*

$$cs(A_1) <_{\mathcal{H}} cs(A_2) \text{ or } cs(A_2) <_{\mathcal{H}} cs(A_1).$$

The FCFS (First-Come, First-served) property is the following: *If* $A_1, A_2 \in E_{\mathcal{H}}$ *are protocol executions,* A_2 *is terminating and*

$$clock(A_1) <_{\mathcal{H}} clock(A_2),$$

then

$$A_1 \text{ is terminating and } cs(A_1) <_{\mathcal{H}} cs(A_2).$$

There are protocols for the critical section problem that satisfy the FCFS property and use much smaller data types than those used by the Bakery Algorithm and its variants. For example Lycklama and Hadzilacos [13] have a protocol for N processes which satisfies the FCFS property and uses registers of data-size 2^4 for each process. It is interesting to investigate the different merits of these protocols and to see if, besides its conceptual simplicity, the Bakery Algorithm has other attractive properties that compensate its use of large data types. One answer has evolved in this investigation. Let us first say it informally. If we have two executions of the protocol in $\mathcal{H}$, it suffices to look at their doorways and at their timestamps in order to find out which enters first the critical section. (The timestamp is the value of $n[i]$ which is written in line l_3.) Even if it is not the case that one of the two doorways precedes another, there is a definable way to tell which of the two has the priority, the rest of the protocol only establishes that the processes proceed in accord with this priority. We will prove that there is a linear ordering relation, $\Rightarrow$, defined on the clock events and extending $<_{\mathcal{H}}$, such that

(1) $\Rightarrow$ is definable from the clock events alone (in $\mathcal{H}$),

(2) If $clock(X) \Rightarrow clock(Y)$, then $cs(X) <_{\mathcal{H}} cs(Y)$.

The definition of $\Rightarrow$ for the Bakery Algorithm (and the Local Clocks protocol) is given in Definition 4.2. (Though this definition is quit simple it will be more meaningful at that later stage.)

Definition 2.3 *A mutual exclusion protocol that has a definable relation $\Rightarrow$ satisfying (1) and (2) above (in each of its higher-level executions) is said to have the "Bakery Definability Property".*

What does it mean that $\Rightarrow$ is definable from the clock events? If $\mathcal{H}_1$ is the substructure obtained from the higher-level view $\mathcal{H}$ defined above, by taking only the clock events and restricting $<_{\mathcal{H}}$ and the predicates and functions of $F_{\mathcal{H}}$ to these clock events, then $\Rightarrow$ is definable in $\mathcal{H}_1$. Namely there is a formula $\varphi(x, y)$ such that for clock events a and b, $\varphi(a, b)$ holds in $\mathcal{H}_1$ iff $a \Rightarrow b$.

It is obvious that in the full higher-level execution $\mathcal{H}$ a linear ordering that satisfy (2) can be easily defined, since the *cs* events are linearly ordered by $<_{\mathcal{H}}$: just define $clock(X) \Rightarrow clock(Y)$ iff $cs(X) <_{\mathcal{H}} cs(Y)$. Our point is that $\Rightarrow$ can be defined by the clock events alone.

3 The higher-level properties of the Bakery Algorithm

Let $\mathcal{S} = (E_{\mathcal{S}}, <_{\mathcal{S}}, F_{\mathcal{S}})$ be a system execution that represents a run of the Bakery Algorithm on N processes in which the register are assumed to be safe, and let ω be the regular return function (see the note following Definition 2.1). The lower-level events of $E_{\mathcal{S}}$ are grouped into a collection $E_{\mathcal{H}}$ of higher-level events, and the functions *clock*, *wait*, and *cs* are defined as in Section 2.2. We will complete now the definition of $\mathcal{H} = (E_{\mathcal{H}}, X_{\mathcal{H}}, <_{\mathcal{H}}, F_{\mathcal{H}})$ by adding to $F_{\mathcal{H}}$ the function ts and the relation *dominates*, defined bellow. Then we will state and prove the crucial higher-level properties of these functions and relations. In Section 4 these properties (called "specifications") will be shown to imply the mutual exclusion and the First Come, First Serve properties. As defined in the previous section, $E_{\mathcal{H}}$ consists of all the *clock*, *wait*, *non-cs*, *cs* and *exit* events, as well as the executions of the protocol (terminating and nonterminating) and the *Initialization* events. Obviously, for every protocol execution, $A \in E_{\mathcal{H}}$,

$$clock(A) <_{\mathcal{H}} wait(A)$$

and for a terminating A

$$wait(A) <_{\mathcal{H}} cs(A).$$

The fact that *clock*(A) is included in A is expressed by

$$begin(A) <_{\mathcal{H}} clock(A) <_{\mathcal{H}} end(A).$$

We now define and add to $F_{\mathcal{H}}$ the function $ts(a)$, the time-stamp of a, defined on *clock* events a as the value written on R_i (in line l_3). $index(A)$ is the id number of the process executing A, i.e., $index(A) = i$ iff A is in P_i.

The following relation is defined on clock events:

$$a \textit{ dominates } b \text{ iff } (ts(a),\ index(a)) > (ts(b),\ index(b)).$$

The linear ordering $>$ used here is the lexicographical ordering:

$$(t, i) > (t', i') \text{ iff } t > t' \text{ or } (t = t' \text{ and } i > i').$$

Obviously, for every clock events a and b with distinct indices:

$$(a \textit{ dominates } b) \text{ or } (b \textit{ dominates } a), \text{ but never both.}$$

The dominance relation is transitive, but it is possible for two distinct clock events to have the same *ts* and the same *index*. It is also possible that $x <_{\mathcal{H}} y$ and x *dominates* y (x has finished its execution and y perhaps read the value ∞ in x's register). Intuitively, if b follows a then b *dominates* a unless a has already terminated; a more carefully formulated property is given in Lemma 3.1(1).

Finally, the function $\Omega(A, j)$ is added to $F_{\mathcal{H}}$. It is defined for terminating executions $A \in E_{\mathcal{H}}$ of the protocol and for indices $j \neq index(A)$. Intuitively,

$\Omega(A,j) = B$ for that $B \in P_j$ that gave A permission to finish the repeat loop of reading R_j (in line $l_5(2)$). Formally, let r be the last read in A of R_j (this last read exists since A is a terminating execution of the protocol, and it successfully passed line l_5 for all $j \neq index(A)$). Let $\omega(r) = w$, and define $\Omega(A,j) = B$ where $B \in P_j$ is the higher-level protocol or *Initialization* event such that $w \in B$.

The main higher-level properties related to Ω and *dominates* are described in the following lemma.

Lemma 3.1 *Let $X \in P_j$, $Y \in P_i$ be protocol executions; set $x = clock(X)$, $y = clock(Y)$.*

(1) *Assume $x <_{\mathcal{H}} y$. Then y dominates x, or X is terminating and $cs(X) <_{\mathcal{H}} end(y)$.*

(2) *Assume $x \not\succ_{\mathcal{H}} y$ and Y is terminating. Then $X \leq_{\mathcal{H}} \Omega(Y,j)$.*

(3) *Assume Y is terminating and $\Omega(Y,j) = X$. Then*

- **(a)** *$x \not\succ_{\mathcal{H}} wait(Y)$.*
- **(b)** *X dominates Y, or X is terminating and $cs(X) <_{\mathcal{H}} end(wait(Y))$.*

Proof: Let us prove (1) first. In case X and Y are in the same process (that is, $i = j$) then $x <_{\mathcal{H}} y$ implies $X <_{\mathcal{H}} Y$, by the seriality of P_i, and hence $cs(X) <_{\mathcal{H}} end(y)$ in this case as well.

So assume $i \neq j$. Then there is in $clock(Y)$ a read, r, of R_j (in the execution of $l_{2.1}$). Let w be the write in x onto R_j (the execution of l_3 in x). As $x <_{\mathcal{H}} y$, $w <_{\mathcal{H}} r$. Since R_j is a safe register, $w \leq_{\mathcal{H}} \omega(r)$. The first case to consider is $w = \omega(r)$, and the second is $w <_{\mathcal{H}} \omega(r)$.

In the first case, $\omega(r) <_{\mathcal{H}} r$ and hence $Value(w) = Value(r) = m$. By definition, this common value m is $ts(x)$. Thus m is the value of $n[j]$ in y and since $ts(y)$ is chosen to be bigger than any of the values in n, $ts(y) > m$. Thus y *dominates* x in this case.

In the second case, $w <_{\mathcal{H}} \omega(r) = w_1$. We will prove that X is terminating and $cs(X) <_{\mathcal{H}} end(y)$ in this case. Since $w <_{\mathcal{H}} w_1$ and w_1 is in process P_j, the structure of the protocol (the fact that there are only two writes onto R_j in the protocol, and w corresponds to the first) implies that w_1 is either the last write in X (the write of ∞ in line l_7) or else $X <_{\mathcal{H}} w_1$. In either case, X is terminating, and $cs(X) <_{\mathcal{H}} w_1$. Since the return function ω is regular, $\omega(r) \not>_{\mathcal{H}} r$. Thus $cs(X) <_{\mathcal{H}} w_1 \not>_{\mathcal{H}} r$, and hence $cs(X) <_{\mathcal{H}} end(y)$ (by Lemma 2.2).

To prove (2) assume that $x \not>_{\mathcal{H}} y$ and Y is terminating. Let a be the first write in X onto register G_j, that is the write of 1 corresponding to line l_1. Since Y is terminating it contains a successful read q of G_j. Namely a read of G_j that returns value 0. As $a <_{\mathcal{H}} x \not>_{\mathcal{H}} y <_{\mathcal{H}} q$, $a <_{\mathcal{H}} q$ by the Russell–Wienner property. Yet a is a write of 1 and q a read of 0 of register G_j, and hence for some write b onto G_j that follows a, $b \not>_{\mathcal{H}} q$. Thus $b_0 \not>_{\mathcal{H}} q$, where b_0 in Y is the write of 0 onto G_j (execution of line l_4). Let r be the last read of R_j in $wait(Y)$. It is the read used in defining $\Omega(Y,j)$. We have $x <_{\mathcal{H}} b_0 \not>_{\mathcal{H}} q <_{\mathcal{H}} r$, and hence $x <_{\mathcal{H}} r$. Thus $\omega(r)$ is either the write on R_j in x or a later write, and in any case $X \leq_{\mathcal{H}} \Omega(Y,j)$.

For the proof of (3), suppose $\Omega(Y,j) = X$, and let r in $wait(Y)$ be the last read of R_j, the one that obtained $(n[j],j) > (n[i],i)$. Then $\omega(r)$ is in X by definition of Ω, and so $\omega(r)$ is either in x or $x <_{\mathcal{H}} \omega(r)$ (in case $\omega(r)$ is the last write in X). The regularity of ω proves that $x \not>_{\mathcal{H}} r$. Thus $x \not>_{\mathcal{H}} wait(Y)$ follows, and this is **(a)**.

To prove (3)**(b)**, assume first that $\omega(r)$ is the write of ∞ in X and thus $x <_{\mathcal{H}} \omega(r)$. In this case X is terminating and $cs(X) <_{\mathcal{H}} \omega(r) \not>_{\mathcal{H}} r$ implies that $cs(x) <_{\mathcal{H}} end(wait(Y))$ (again by Lemma 2.2).

Now assume that $\omega(r)$ is the write of the timestamp in x. We shall analyze the two possibilities for $\omega(r)$: $\omega(r) <_{\mathcal{H}} r$, and $\omega(r)$ overlapping r. Assume first $\omega(r) <_{\mathcal{H}} r$, and thus $Value(r) = Value(\omega(r))$ by the definition of safeness. Then x *dominates* y because the read r was defined to return a value that satisfies the until condition of l_5 and because this value is the timestamp of X.

Now assume that $\omega(r)$ and r overlap, that is $\omega(r) \not>_{\mathcal{H}} r$ and $r \not>_{\mathcal{H}} \omega(r)$. We will prove that x *dominates* y in this case. Let a and b be the writes onto G_j of the values 1 and 0 respectively in X. Thus $a <_{\mathcal{H}} \omega(r) <_{\mathcal{H}} b$. Let q be the read of G_j in Y that returned the value 0 (since Y is terminating it contains such a read in $l_5(1)$). Then $q <_{\mathcal{H}} r$. Since $q <_{\mathcal{H}} r \not>_{\mathcal{H}} \omega(r) <_{\mathcal{H}} b$, $q <_{\mathcal{H}} b$ follows. Thus $a <_{\mathcal{H}} q$ would imply that q returns 1 which is not the case. Hence $q \not>_{\mathcal{H}} a$. It easily follows in this case that the read of R_i in x obtains the value of the timestamp written by y (because it strictly lies in between the write of the timestamp in y and the write of ∞). Hence x *dominates* y.

4 The higher-level specifications and consequences

We begin by restating the the higher-level properties that hold for the Bakery Algorithm, but now in an abstract form, for an arbitrary system execution, $\mathcal{H}$, not necessarily connected with a protocol. Then we will define $\Rightarrow$ and prove its properties from these higher-level specifications.

First we define the signature of our structures, which is a list of sorts, predicates, functions, and constants. Defining the signature is defining the language used to discuss the higher-level Bakery properties. Then we will define the Bakery Specifications, which are the first-order properties expressed in the language just formulated, and we will prove that these specifications imply the required correcness properties of the protocol. At least in theory there is a strict separation between the signature and the properties expressed by its language. However, in practice, there may be a need for a less rigid style of presentation in which the signature definition is accompanied with motivating explanations. Formaly speaking, these belong to the specification part, and so we write them in square parenthesis.

The Bakery Signature

Sorts: The set of events is denoted E. The set of atemporal objects is denoted X. (But if $\mathcal{H}$ is any system execution that interprets this signature, then we will write $E_{\mathcal{H}}$ and $X_{\mathcal{H}}$ for the interpretation of these sorts.) [The atemporal objects are just the indices $\{1, \ldots, N\}$ of the processes. Each $1 \leq i \leq N$ is a constant.]

Any system execution has a precedence relation $<$ defined on the set of events E, and a unary predicate *Terminating* defined on E. The Russell–Wiener property and the Lamport Finiteness Property are assumed for these relations.

Functions There is a function $index : E \longrightarrow X$. (For every event $e \in E$, $index(e) = i$ indicates that e is executed by process P_i.) There are additional functions $clock, wait, cs, non-cs$ that are defined on the protocol events in E (see below) and return events (further explanations are given below).

The function Ω is define. [The value $\Omega(Y, j)$ is defined whenever Y is a terminating protocol event and $j \neq index(Y)$ is an index. In this case $\Omega(Y, j)$ is in P_j either a protocol event or an initialization event.]

Predicates Predicate $P_i(e)$ may hold (or not hold) for events e and process index i. [For every process index $i \in X = \{1, \ldots, N\}$, and event $e \in E$, $P_i(e)$ indicates that e is executed by process P_i. So $P_i(e)$ holds iff $index(e) = i$. Hence P_i is dispensible, but we find it convenient to use it. For greater flexibility we may also write $e \in P_i$ instead of $P_i(e)$. So the sets P_i form a partition of E.]

A unary predicate *protocol* is defined on E. (If $protocol(e)$ holds, then e is said to be a protocol execution, although this signature and specification is detached from any specific protocol, we keep the name “protocol execution" to refer to higher-level events that represent executions of the protocol.)

Another predicate on E is *Initialization*. [If *Initialization*(e), then e is said to be an initial event. We assume that each P_i has a unique initial event and that the initial events precede any other event.]

Other unary predicates on E are: *non-cs*, *clock*-type, *wait*-type, and *cs*-type. We just say that e is a *clock* event, instead of writing *clock*-type(e) etc. [These predicates partition the set of event: any event is either an initialization event, or a protocol execution, or a non-cs execution, or a clock event, or a wait event, or a cs event. The intuition that any clock event is part of a protocol execution is expressed by the function *clock* which is defined on the protocol executions. If p is a protocol execution, then $clock(p)$ is its clock part. Similarly, $wait(p)$ is defined. The function $cs(p)$ is only defined on terminating protocol executions p, since non-terminating executions have no cs part. Thus the function *cs* is defined on terminating protocol events and is onto the *cs* class of events.]

There is a binary relation *dominates* defined on E. Actually it is defined on the *clock* events.

A system execution $\mathcal{H} = (E_{\mathcal{H}}, X_{\mathcal{H}};\ <_{\mathcal{H}}\ , F_{\mathcal{H}})$ that interprets the Bakery Signature is defined to satisfies the Bakery Specifications iff the following holds in $\mathcal{H}$:

The Bakery Specifications

For each P_i, the protocol events in P_i are linearly ordered.

We assume initialization events; one in each process. An initialization event precedes any non initialization event. For notational convenience we assume that all the functions are defined on the initialization events as well, and are the identity there ($clock(X) = X$, for example, for an initialization event X).

1. For every protocol event X

$$begin(X) <_{\mathcal{H}} clock(X) <_{\mathcal{H}} wait(X) <_{\mathcal{H}} end(X).$$

If X is a terminating protocol event, then $cs(X)$ is defined and

$$wait(X) <_{\mathcal{H}} cs(X) <_{\mathcal{H}} end(X).$$

(See Subsection 2.2 for the definition of $cs(X) <_{\mathcal{H}} end(X)$.)

2. There is a binary relation called *dominates* and defined on clock events. For any clock events x and y with different indices,

$$x \textit{ dominates } y \text{ or } y \textit{ dominates } x, \text{ but not both.}$$

We may write X *dominates* Y instead of $clock(X)$ *dominates* $clock(Y)$.

3. If $x <_{\mathcal{H}} y$ are *clock* events, and $x = clock(X)$, then y *dominates* x, or the protocol event X is terminating and $cs(X) <_{\mathcal{H}} end(y)$.

4. The function Ω is such that for any terminating protocol event, Y, and index $j \neq$index(Y), $\Omega(Y, j) = X$ is a protocol event (or the initialization event) of P_j.

5. Assume that $X \in P_j$ and $Y \in P_i$ are protocol events, $x = clock(X)$, and $y = clock(Y)$. If Y is terminating then:

   (a) If $x \not>_{\mathcal{H}} y$ then $X \leq_{\mathcal{H}} \Omega(Y, j)$.

   (b) If $X = \Omega(Y, j)$, then

      i. $clock(X) \not>_{\mathcal{H}} wait(Y)$.

      ii. X *dominates* Y or
      X is terminating and $cs(X) <_{\mathcal{H}} end(wait(Y))$.

6. The transitivity of $\Rightarrow$. Define the relation $x \Rightarrow y$ on the set of clock events by $x \Rightarrow y$ iff

$$x \neq y,\ x \not\succcurlyeq_{\mathcal{H}} y \text{ and either (1) } x <_{\mathcal{H}} y \text{ or (2) } y \textit{ dominates } x.$$

   Then $\Rightarrow$ is a linear ordering on the *clock*-type events.

This ends the Bakery Specifications.

Except for item (6), we have seen that these specifications hold for the Bakery Algorithm (Lemma 3.1). We are going to show now that (6) holds as well, by proving that it is a consequence of (1)—(5) and the transitivity of the dominance relation (which is obvious for the Bakery Algorithm and the Local Clocks protocol but is not satisfied by the Bounded Protocol of Section 5). First we need a lemma.

Lemma 4.1 *Assume that system execution $\mathcal{H}$ satisfies the higher-level Bakery Specifications (1)—(5). For any two protocol events, $A, B \in E_{\mathcal{H}}$, with distinct indices, if*

1. *$clock(A) \not\succcurlyeq_{\mathcal{H}} clock(B)$,*

2. *B dominates A,*

3. *B is terminating,*

then A is terminating and $cs(A) <_{\mathcal{H}} end(wait(B))$.

Proof: Say $A \in P_j$ and $B \in P_i$ and $i \neq j$. Since B is terminating, $\Omega(B, j)$ is defined and $A \leq_{\mathcal{S}} \Omega(B, j)$ by (5)(a).
Case 1: $A = \Omega(B, j)$. Since B *dominates* A, it is not the case that A *dominates* B, and hence 5(b)(ii) implies that

$$A \text{ is terminating and } cs(A) <_{\mathcal{H}} end(wait(B)).$$

Case 2: $A <_{\mathcal{H}} \Omega(B, j)$. Then A is terminating (because $X <_{\mathcal{H}} Y$ always implies that X is terminating). Since $\Omega(B, j) \not\succcurlyeq_{\mathcal{H}} end(wait(B))$,

$$cs(A) <_{\mathcal{H}} end(wait(B))$$

follows.

Let us repeat the definition of $\Rightarrow$. For any system execution $\mathcal{H}$ for the higher-level Bakery signature, the relation $x \Rightarrow y$ on the *clock* events was defined as follows:

Definition 4.2 *$x \Rightarrow y$ iff $x \neq y, x \not\twoheadrightarrow_{\mathcal{H}} y$ and either*
(1) $x <_{\mathcal{H}} y$ or
(2) y dominates x.

In words, $\Rightarrow$ is the relation extending $<_{\mathcal{H}}$ such that in case x and y are $<_{\mathcal{H}}$ incomparable, then $x \Rightarrow y$ is determined on the basis of dominance.

If $x <_{\mathcal{H}} y$, we say the reason for $x \Rightarrow y$ is 'precedence', and, otherwise, if $y \not\twoheadrightarrow_{\mathcal{H}} x$ and y *dominates* x, then we say that the reason is 'dominance'.

Observe that $\Rightarrow$ is definable in the restriction of $\mathcal{H}$ to the *clock* events.

Corollary 4.3 *Assume $\mathcal{H}$ satisfies the first five higher-level Bakery Specifications, and suppose that 'x dominates y' is a transitive relation, then the relation $\Rightarrow$ is a linear ordering (extending $<_{\mathcal{H}}$) on the clock events.*

Proof: Clearly $\Rightarrow$ extends $<_{\mathcal{H}}$ on the clock event, for if $x <_{\mathcal{H}} y$ then $x \not\twoheadrightarrow_{\mathcal{H}} y$. It follows from the seriality of each process and item (2) of the Bakery Specifications that for any two distinct *clock* events either $x \Rightarrow y$ or $y \Rightarrow x$, but not both. To prove transitivity, suppose, towards a contradiction, that $x \Rightarrow y$, $y \Rightarrow z$ but not $x \Rightarrow z$. Hence $z \Rightarrow x$ (as $x = z$ is not possible).

Case 1 The reason for all of $x \Rightarrow y$, $y \Rightarrow z$, and $z \Rightarrow x$ is dominance. Then y *dominates* x, z *dominates* y, and x *dominates* z. The transitivity of dominance implies that z *dominates* x, but this contradicts the antisymmetry of dominance.

Case 2 For two out of the three relations the reason is precedence. For example, suppose $x <_{\mathcal{H}} y$ and $y <_{\mathcal{H}} z$. Then $x <_{\mathcal{H}} z$ as well, and hence $x \Rightarrow z$ (in contradiction to $z \Rightarrow x$).

Case 3 Only for one of the three relations the reason precedence, and it is dominance for the other two. For example, suppose that $x <_{\mathcal{H}} y$, but both of y and z , and x and z, are $<_{\mathcal{H}}$ incomparable (and hence have distinct indices). Then, since $y \Rightarrow z$ and $z \Rightarrow x$,

$$z \textit{ dominates } y \text{ and } x \textit{ dominates } z.$$

By the transitivity of dominance, x *dominates* y.

Let X, Y, and Z be protocol events such that $x = \mathit{clock}(X)$, $y = \mathit{clock}(Y)$, $z = \mathit{clock}(Z)$. As $x <_{\mathcal{H}} y$, by item (3), either y *dominates* x or X is terminating and $cs(X) <_{\mathcal{H}} \mathit{end}(y)$. Since it is not the case that y *dominates* x, X is terminating and

$$cs(X) <_{\mathcal{H}} \mathit{end}(y). \tag{1}$$

Since x and z are $<_{\mathcal{H}}$ incomparable, $z \not>_{\mathcal{H}} x$. So Lemma 4.1 can be applied (X is terminating) to conclude that

$$Z \text{ is terminating and } cs(Z) <_{\mathcal{H}} \mathit{end}(\mathit{wait}(X)). \tag{2}$$

Hence $cs(Z) <_{\mathcal{H}} cs(X)$, and so $cs(Z) <_{\mathcal{H}} \mathit{end}(y)$.
The lemma is applied again, to Y and Z (which is now known to be terminating). Since $y \not>_{\mathcal{H}} z$, we conclude that

$$Y \text{ is terminating and } cs(Y) <_{\mathcal{H}} \mathit{end}(\mathit{wait}(Z)). \tag{3}$$

Hence $\mathit{end}(y) <_{\mathcal{H}} cs(Z)$. Thus $cs(Z) <_{\mathcal{H}} cs(Z)$ which is impossible.

This proves the corollary, and hence item 6 of the Specifications holds for the Bakery Algorithm (since we have shown the first five items, and as the dominance relation, as defined for the Bakery Algorithm, is clearly transitive).

The mutual exclusion and the FCFS properties are consequences of the following lemma and the fact that $\Rightarrow$ is a linear ordering extending $<_{\mathcal{H}}$. This lemma establishes the Bakery Definability Property 2.3.

Lemma 4.4 *Assume the higher-level Bakery Specifications (1)—(6) for $\mathcal{H}$. Assume X and Y are protocol events with $\mathit{clock}(X) \Rightarrow \mathit{clock}(Y)$. If Y is terminating then X is terminating and $cs(X) <_{\mathcal{H}} cs(Y)$.*

Proof: The case where X and Y have the same index is immediate (because then $X <_{\mathcal{H}} Y$ since $\Rightarrow$ extends $<_{\mathcal{H}}$). So assume that the indices are distinct. Put $x = clock(X)$, $i = index(X)$, $y = clock(Y)$, and let $Z = \Omega(Y, i)$. Z is a protocol event in P_i (or the initialization). The definition of $x \Rightarrow y$ requires that $x \not\geqslant_{\mathcal{H}} y$. Hence $X \leq_{\mathcal{H}} \Omega(Y, i)$ by (5)(a).

Case 1 $X <_{\mathcal{H}} Z$. Then X is terminating. By (5)(**b**), $clock(Z) \not\geqslant_{\mathcal{H}} wait(Y)$, and as $X <_{\mathcal{H}} clock(Z)$ in this case, and $wait(Y) <_{\mathcal{H}} cs(Y)$, $cs(X) <_{\mathcal{H}} cs(Y)$ follows.

Case 2 $X = \Omega(Y, i)$. By (5(**a**)) either

(a) x dominates y or

(b) X is terminating and $cs(X) <_{\mathcal{H}} end(wait(Y))$.

If (b) holds, then $cs(X) <_{\mathcal{H}} cs(Y)$ follows from $wait(Y) <_{\mathcal{H}} cs(Y)$. So assume that x *dominates* y (and hence it is not the case that y *dominates* x). But $x \Rightarrow y$ and the negation of y *dominates* x, imply that $x <_{\mathcal{H}} y$ (by the definition of $\Rightarrow$). It follows from item 3 of the Bakery Specifications that X is terminating and $cs(X) <_{\mathcal{H}} end(clock(Y))$. Hence $cs(X) <_{\mathcal{H}} cs(Y)$. ∎

Liveness properties. The Bakery Specifications do not suffice to prove the loveness properties of the protocol. Additional specifications are required in order to give a unified higher-level proof for the liveness properties of the protocols presented here. This will be done at the end of the following section. In that section we deal with the Bounded Protocol, prove that it satisfies the Bakery Specifications, and then formulate and prove its liveness property. This proof is general enough to apply both to the Bakery Protocol and to the Local Clocks Protocol.

5 The bounded protocol

This is the main section of the paper, it contains a version of the Bakery Algorithm that uses only bounded values (the bound depends on the number of processes). The registers are assumed to be atomic (serial, for simplicity).

We begin with a description of the different data types, local variables and registers used by the processes executing the bounded protocol.

The kth execution of the protocol (by a process P_i, $1 \le i \le N$) is said to be of phase $K = 0, 1, 2$ if $k = K \pmod 3$. Thus, the first execution is of phase 1, the third of phase 0, etc. The numbers $\{0, 1, 2\}$ are accordingly called 'phase numbers'. There is a local variable $v.phase$ which holds numbers in the range $0, 1, 2$ and which is increased by $1 \pmod 3$ to reflect the new phase at the beginning of each execution of the protocol (see line 1 of the protocol). The initial value of $v.phase$ is 0. Actually $v.phase$ is a field in a local variable v which contains two other fields: *status* and *evidence*. The variable v is written onto register R_i (three times, at lines 2, 4 and 8). The variable $v.status$ is boolean: the value 0 indicates that the process is inactive, and the value 1 indicates that the process begins an execution of the protocol. See line 1 where $v.status$ is set to 1, and line 7 where it is reset to 0 before the final write.

The variable $v.evidence$ carries values of type called evidence-type which replaces the unbounded timestamps. An evidence-type is either the special 'undefined' value $*$ or else an array, a, of the form $(a[1], \ldots, a[i])$ where each $a[i]$ is a phase number. There are thus $1 + 3^i$ possible values in evidence-type. In case a is not $*$ (and then, for all j, $a[j]$ is a a phase number), a is called a 'phase vector'.

The relation x *dominates* y is defined first for phase numbers $x, y \in \{0, 1, 2\}$ whenever $x = y + 1 \pmod 3$. Thus 0 *dominates* 2, for example. For any two distinct $x, y \in \{0, 1, 2\}$ either x *dominates* y or y *dominates* x, but not both.

Definition 5.1 *For phase vectors a and b of lengths i and k respectively, define a dominates b iff*

(1) there is $j \le \min\{i, k\}$ such that $a[j] \ne b[j]$ and, for the first such j, $a[j]$ dominates $b[j]$, or

(2) $k < i$ and, for every $j \leq k$, $a[j] = b[j]$.

This dominance relation is used in line 5 of the protocol. (Observe that it is possible for the value of $r_j.evidence$ to be $*$, and in such a case $r_j.evidence$ *dominates* $v.evidence$ will not hold of course.)

The initial value of R_i is assumed to be the value v determined by $v.status = 0$, $v.phase = 0$, $v.evidence = *$. The registers R_i are assumed to be serial (that is, the read/write events on R_i are linearly ordered in $<_S$).

The protocol can now be given.

Bounded Protocol for P_i $(1 \leq i \leq N)$

1. $v.evidence := *;\ v.status := 1;\ v.phase := v.phase + 1 \pmod 3$
2. write(R_i, v)
3.
   (a) for $j := 1$ to $i - 1$ do (read(R_j, r_j); $v.evidence[j] := r_j.phase$)

   (b) $v.evidence[i] := v.phase$
4. write(R_i, v)
5. forall $j \neq i$
   repeat read(R_j, r_j)
   until $r_j.status = 0$ or $r_j.evidence$ *dominates* $v.evidence$
6. Critical Section
7. $v.status := 0$
8. write(R_i, v).

Lines 1–4 are the doorway, and lines 3 and 4 constitute the *clock*. Line 5 is the wait part of the protocol, line 6 is the critical section, and lines 7,8 are the exit.

Notice that the for loop in line 3 of the protocol is done in increasing order, and that the protocol is not symmetric. Process P_i only reads the registers of the lower processes P_j for $j < i$. In case $i = 1$, this reading loop has an empty execution of course.

We wish now to prove that the Bakery Specifications of Section 4 hold for this bounded protocol as well. So let $\mathcal{S}$ be a system execution in which $P_1, \ldots, P_N$ execute the protocol, and let $\mathcal{H}$ be the corresponding higher-level view, in which protocol executions, as well as non-critical-section, clock, wait, critical section, and exit events are grouped into higher-level events. Assume in $\mathcal{S}$ a return function ω that associates with each read r of the serial register R_i the last write $\omega(r)$ (in P_i) such that $\omega(r) <_{\mathcal{S}} r$. (Assume an initial write onto R_i that precedes all read events.)

First define the function *collect* on clock events. Given a clock event $e \in E_{\mathcal{H}}$ with $i = \mathit{index}(e)$, $\mathit{collect}(e)$ is an array of length i of protocol executions (and initial events) in $E_{\mathcal{S}}$: $\mathit{collect}(e)[\ell] \in P_l$, for $1 \leq \ell \leq i$, is defined as follows. For the highest index $\ell = i$ define $\mathit{collect}(e)[i] = E$ where E is the protocol execution such that $e = \mathit{clock}(E)$. Now, for $\ell < i$, let $r \in e$ be the read event of register R_ℓ, corresponding to the instruction in line 3. Then $\omega(r)$ is a write onto R_ℓ (done by P_ℓ). Find the prtocol execution, or the initialization event, $W \in P_\ell$ such that $\omega(r) \in W$, and then define $\mathit{collect}(e)[\ell] = W$.

For any execution $W \in E_{\mathcal{H}}$ of the protocol, we denote with $\mathit{phase}(W)$ the value of variable $v.\mathit{phase}$ after the execution of line 1 (and for an initialization event W, define $\mathit{phase}(W) = 0$). Clearly $\mathit{phase}(W)$ is a phase number. If $\langle W_1 \ldots W_i \rangle = \overline{W}$ is a vector of protocol executions and initialization events, then $\mathit{phase}(\overline{W})$ denotes the vector $\langle \mathit{phase}(W_1), \ldots, \mathit{phase}(W_i) \rangle$. For a clock event e, $\mathit{evidence}(e)$ is the value of variable $v.\mathit{evidence}$ in e after executing line 3 (it is a phase vector of length $i = \mathit{index}(W)$). It follows immediately that if e is a clock event, then

$$\mathit{evidence}(e) = \mathit{phase}(\mathit{collect}(e)).$$

Next we define the dominance relation and the function Ω. Given two clock events, e_1 and e_2 we say that e_2 *dominates* e_1 iff

$$evidence(e_2)\ dominates\ evidence(e_1).$$

(See Definition 6.1.) Dominance is not transitive of course, but, as we shall see, the derived relation $\Rightarrow$ *is* transitive, and the specifications do not require more.

The function $\Omega(X, j)$ is defined for any terminating protocol event X and index $j \neq index(X)$, by the same definition given for the Bakery Algorithm. That is, $\Omega(X, j) \in P_j$ is the higher-level protocol event (or *Initialization* event) that contains $\omega(r)$, where $r \in X$ is the last read of R_j (corresponding to line 5). Observe that if r is the last read of register R_j, in $wait(X)$, then the write $\omega(r)$ is never an execution of line 2. (Because $r_j.status = 1$, and $r_j.evidence = *$ cannot satisfy the `until` condition of line 5.)

Except perhaps for items (3) and (6), the proof that the Bakery Specifications hold for this protocol is quite simple and resembles the corresponding proof given for Lemma 3.1. So I assume that the reader has already checked items (1), (2), (4) and (5). In fact, since the registers are assumed to be serial, property (5)(**b**)(i) can be strengthened to:
(5′)(**b**)(i) If $\Omega(Y, i) = X$ then $clock(X) <_{\mathcal{H}} end(wait(Y))$.

We are going to define a linear ordering $\prec$ on the clock events and prove three lemmas that imply items (3) and (5) of the Bakery Specifications.

Definition 5.2 *Given clock events e_1 and e_2, of indices i_1 and i_2 respectively, define $e_1 \prec e_2$ iff $e_1 \neq e_2$ and either*
(1) For some $j \leq \min\{i_1, i_2\}$ $collect(e_1)[j] \neq collect(e_2)[j]$ and, for the first such j, $collect(e_1)[j] <_{\mathcal{H}} collect(e_2)[j]$, or
(2) $i_1 < i_2$ and for all $j \leq i_1$, $collect(e_1)[j] = collect(e_2)[j]$.

It can be seen that $\prec$ is a linear ordering that extends $<_{\mathcal{H}}$ on the clock events. We will prove that $\prec$ is equal to the $\Rightarrow$ relation of Definition 4.2, thereby proving that $\Rightarrow$ is transitive (this is item (5)).

Lemma 5.3 *Let $X \in P_i$ and $Z \in P_\ell$ be protocol executions, where $i \geq \ell$. Let $x = clock(X)$, and $z = clock(Z)$. If $collect(x)[\ell] <_{\mathcal{H}} Z$, then $x \prec z$. If, moreover, Z is terminating, and $i > \ell$, then $X \leq_{\mathcal{H}} \Omega(Z, i)$.*

Proof: If $i = \ell$ then $collect(x)[\ell] = X$, which implies $X <_{\mathcal{H}} Z$, and the lemma is obvious. So assume $i > \ell$. Put $W = collect(x)[\ell]$, and assume $W <_{\mathcal{H}} Z$. Let r be the read in x of register R_ℓ. Then $\omega(r)$ is in W, and hence (as the registers are serial) $r <_{\mathcal{S}} a$, where a is the first write in Z onto R_ℓ (done for line 2). Hence $r <_{\mathcal{S}} s$ for every read s in Z. Since r is after w_0, the first write in X, the 'moreover' clause of the lemma can now be deduced from the fact that no read in Z can obtain a write that precedes w_0.

To prove that $x \prec z$, observe that the execution of the reads in line 3 is in ascending order from 1 to $i-1$. So, as $\ell < i$, if $j < \ell$ then for every read, u, of R_j, in x, and read s of R_j in z, $u <_{\mathcal{S}} r <_{\mathcal{S}} s$. This implies that $collect(x)[j] \leq_{\mathcal{H}} collect(z)[j]$ for $j < \ell$. As $collect(x)[\ell] = W <_{\mathcal{H}} Z$, and $Z = collect(z)[\ell]$, $x \prec z$.

Lemma 5.4 *Suppose that $x = clock(X)$, $y = clock(Y)$. If X is terminating and $cs(X) <_{\mathcal{H}} end(y)$, then $x <_{\mathcal{H}} y$.*

Proof: The case where X and Y have the same index is easy and then $X <_{\mathcal{H}} Y$ holds. (Indeed, otherwise, $Y \leq_{\mathcal{H}} X$ implies that $y \leq_{\mathcal{H}} x$, and as $cs(X) <_{\mathcal{H}} end(y)$, $cs(X) <_{\mathcal{H}} end(x)$ follows, and then $clock(X) <_{\mathcal{H}} cs(X)$ implies $cs(X) <_{\mathcal{H}} cs(X)$ which is impossible.)

Suppose $j = index(Y) \neq index(X)$. Put $Z = \Omega(X, j)$, and $z = clock(Z)$. Then by the stronger property (5')(**b**)(i), $z <_{\mathcal{H}} end(wait(X))$ holds. So, as $wait(X) <_{\mathcal{H}} cs(X)$, $z <_{\mathcal{H}} cs(X)$. Since $\neg(yHsacs(x))$ (which follows from $cs(x) <_{\mathcal{H}} end(y)$), $z <_{\mathcal{H}} y$ holds. This implies that $Z <_{\mathcal{H}} Y$. Now $x <_{\mathcal{H}} y$ follows, because otherwise $y \not>_{\mathcal{H}} x$, and then property 5(a) of the Bakery Specification would imply that $Y \leq_{\mathcal{H}} \Omega(X, j) = Z$.

Lemma 5.5 *If $x = clock(X)$, $y = clock(Y)$, if $x \prec y$ then:*

1. *y dominates x or else*

2. *X is terminating, and $cs(X) <_{\mathcal{H}} end(y)$. (Hence, by the previous lemma, $x <_{\mathcal{H}} y$ can also be concluded in this case.)*

Proof: Say $i = index(X)$. We prove the lemma by induction on $j = index(Y)$, and for each j, by induction on the cardinality of the finite set $\{a \in P_j \mid a <_{\mathcal{H}} Y\}$.

In case $i = j$ the seriality of P_i implies our lemma immediately, because $\prec$ extends $<_{\mathcal{H}}$. So assume $i \neq j$. Looking at the definition of $x \prec y$ (Definition 6.2), there are two cases. We begin with the second which is easier.

Case 2: $i < j$ and $collect(x)$ equals the restriction of $collect(y)$ to the first i values. As $evidence(u) = phase(collect(u))$ for every clock event u, $evidence(x)$ equals the restriction of $evidence(y)$ to the first i values. By definition, as $i < j$, this implies that y *dominates* x. Hence 1 hold nd the lemma is proved in this case.

Case 1: Not Case 2. Denoting $W_x^n = collect(x)[n]$, $W_y^n = collect(y)[n]$, we have that $W_x^\ell <_{\mathcal{H}} W_y^\ell$ for the first index ℓ such that $\ell \leq \min\{i,j\}$ and $W_x^\ell \neq W_y^\ell$. In case $\ell = i$, $X = W_x^\ell <_{\mathcal{H}} W_y^\ell = collect(y)[\ell]$ implies that X is terminating, and $X <_{\mathcal{H}} collect(y)[\ell] \not>_{\mathcal{H}} y$ imply that $X <_{\mathcal{H}} end(y)$ (by Lemma 2.2). Hence $cs(X) <_{\mathcal{H}} end(y)$. Thus we may assume

$$\ell < i.$$

Put $p = phase(W_x^\ell), q = phase(W_y^\ell)$. If $p + 1 = q \pmod 3$, then

$$evidence(y) \ dominates \ evidence(x),$$

and (by definition) y *dominates* x.

Now assume that $p + 1 \neq q \pmod 3$. We will see in this case that X is terminating, and $cs(X) <_{\mathcal{H}} end(y)$. Denote by W^+ the successor of W_x^ℓ in P_l, i.e., the protocol execution by P_ℓ immediately following W_x^ℓ (W^+

exists because $W_x^\ell <_{\mathcal{H}} W_y^\ell$). Since $phase(W^+) = phase(W_x^\ell) + 1 \pmod 3$, $W^+ \neq W_y^\ell$ and hence

$$W^+ <_{\mathcal{H}} W_y^\ell.$$

Apply Lemma 5.3 to X and W^+ (as Z) to conclude

$$x \prec w^+ = clock(W^+), \text{ and } X \leq_{\mathcal{H}} \Omega(W^+, i).$$

This last relation implies that either

$$X \ dominates \ W^+$$

or

$$X \text{ is terminating, and } cs(X) <_{\mathcal{H}} end(wait(W^+)).$$

To see this, use item (5)(**b**)(ii) of the Bakery Specifications in case that $X = \Omega(W^+, i)$, and, in case $X <_{\mathcal{H}} \Omega(W^+, i)$, put $V = \Omega(W^+, i)$ and use item (4)**(a)** to deduce $clock(V) \not\succ_{\mathcal{H}} wait(W^+)$, and then $X <_{\mathcal{H}} V$ implies $X <_{\mathcal{H}} end(wait(W^+))$ (by Lemma 2.2).

Now, since $l \leq \min\{i, j\}$ and $W^+ <_{\mathcal{H}} W_y^l$, we may apply the inductive hypothesis to x and w^+ (we showed that $x \prec w^+$) to conclude that either

$$w^+ dominates \ x$$

or

$$X \text{ is terminating and } cs(X) <_{\mathcal{H}} end(w^+).$$

Since either w^+ *dominates* x is false, or x *dominates* w^+ is false, the last two displayed statements imply that X is terminating, and $cs(X) <_{\mathcal{H}} end(W^+)$. But $W_y^l = collect(y)[l] \not\succ_{\mathcal{H}} y$, and $W^+ <_{\mathcal{H}} W_y^l$ and hence $W^+ <_{\mathcal{H}} end(y)$. Hence $cs(X) <_{\mathcal{H}} end(y)$. ∎

Now we can complete the proof of the Bakery Specifications. To prove item (3) assume that $x <_{\mathcal{H}} y$ are clock events. Since $\prec$ extends $<_{\mathcal{H}}$, $x \prec y$ follows. Lemma 5.5 can be applied to give exactly (3).

To prove the transitivity of $\Rightarrow$, we use Lemma 5.5 to show that $x \prec y$ iff $x \Rightarrow y$. In fact it suffices to prove only one direction since $\prec$ is a linear ordering and as for distinct x and y, either $x \Rightarrow y$ or $y \Rightarrow x$, but not both. So assume $x \prec y$. Then $x \not\succ_{\mathcal{H}} y$ because $\prec$ extends $<_{\mathcal{H}}$. If it is not the case that y *dominates* x, then Lemma 5.5 implies that $x <_{\mathcal{H}} y$.

Since $\prec$ is transitive, item (5) of the specifications follows.

5.1 The deadlock freedom

Informally, a *deadlock* is a state of the system execution that does not evolve to the next desirable stage. For the mutual exclusion problem, a desirable state is one in which there is a process in its critical section. Thus the system is deadlocked if despite its activity none of its processes will access their critical sections. A particular process is *locked out* if this process cannot proceed to its next stage (but other processes may). It is well known and easy to see that (for finitely many processes) the lockout freedom is a consequence of the deadlock freedom and the FCFS property. In this subsection we prove the deadlock freedom for the bounded protocol. For this, we return to the protocol itself and analyze its wait loop.

Formally (for the mutual exclusion problem), a *deadlock* in $\mathcal{H}$ is the following statement: *there are only finitely many protocol events in $E_{\mathcal{H}}$, and one of them (at least) is nonterminating.*

By definition, a higher-level event is terminating iff it is a finite collection of terminating lower-level events. If X is a nonterminating protocol execution, then it must contain infinitely many read events (in $wait(X)$) because all lower-level events are assumed to be terminating. We rely on the following property.

Lemma 5.6 *If $X \in P_i$ is a nonterminating protocol event, then there is an index $j \neq i$ such that one of the following alternatives holds for P_j:*

1. *P_j contains infinitely many protocol events (necessarily terminating).*

2. *P_j contains a nonterminating protocol event Y and $X\,dominates Y$.*

Proof. Since X is nonterminating, it contains in $wait(X)$ infinitely many reads of some register R_j. Assume that (1) does not hold and there is only a finite number of protocol events in P_j. There are thence only a finite number of write events on register R_j. If the last write on R_j is the initial write or the last write (corresponding to line 8), then any read of R_j executed afterwards in X will terminate the `repeat` loop of line 5 (since the initial value of field *status* as well as the value determined by line 8 are 0). Since X is assumed to be nonterminating, the last write on R_j corresponds to line 4. Then it is necessarily the case that P_j contains a nonterminating protocol event Y in which this last write on R_j occurs. Consider now any read of R_j done in X after that last write on R_j, and conclude that $X\,dominates Y$.

The reader should check that this lemma holds for the previous protocols (with unbounded values) and we adopt it as the required higher-level Liveness Specification property. ∎

Our aim now is to show how the statement of this lemma immplies (together with the Bakery Specifications 1–6) that there is no deadlock.

Assume a deadlock in $\mathcal{H}$, and let $L \subset E_{\mathcal{H}}$ be the set of all nonterminating protocol events. By assumption L is non-empty, and it is finite since each of the serial processes can contain at most one nonterminating event.

Let $X \in L$ be the $\Rightarrow$ minimal element in L (it exists since $\Rightarrow$ is a linear ordering). Since X is nonterminating and as there is only a finite number of protocol events in our execution, it follows from the Liveness Specification property that there exists a nonterminating protocol execution Y such that

$$X\,dominates Y.$$

Consider $x = clock(X)$ and $y = clock(Y)$. If $x <_{\mathcal{H}} y$, then the Bakery Specification property 3 and the fact that X is nonterminating imply that $y\,dominates x$. This however is not the case (by assumption) and hence $y \not>_{\mathcal{H}} x$. By definition this implies that $y \Rightarrow x$. But this is again a contradiction, and hence there is no deadlock.

6 Conclusions

Lamport [10] describes the limitations of assertional proof techniques and advocates his new approach. Assertional techniques he writes are satisfactory for proving the correctness of individual subroutines (for which the correctness conditions can be expressed in terms of objects used directly by the programs), but for large programs "one would like the correctness conditions to be stated in terms of higher-level concepts". I believe we have gone here one step further in this direction: not only are the correctness conditions *stated* in a higher-level language, but are even *proved* in higher-level terms that involve no lower level textual reference (in the second stage of the proof).

The proofs given here are clearly decomposed into two stages involving following steps.

$$\text{[The program]} \rightarrow \left[\begin{array}{c}\text{Higher-level specifications}\\ \text{(the Bakery Specifications)}\end{array}\right] \rightarrow \left[\begin{array}{c}\text{Correctness conditions}\\ \text{(Mutual Exclusion etc.)}\end{array}\right]$$

The intermediate higher-level specifications proved in the first stage are quite simple to state and to prove. Our proof at this stage relied on the reader's intuition, and, for a rigorous proof relating the program text to the resulting events, one must use an assertional or any other formal method. It is the second stage however, in which no program text is involved, that interests us most in this paper. The statements proved at this stage are about system executions: If $\mathcal{H}$ satisfies the higher-level (bakery) specifications then it necessarily satisfies the correctness conditions. It is this stage that reflects the main ideas and intentions of the designer of the protocol. Since it deals with $\Rightarrow$ and higher-level global statements that are unrelated to the program text, an assertional method is probably inappropriate for this stage. My monograph [2] presents in details and more formally this approach of relying on system-executions to argue about higher-level properties of protocols.

The desire to present in a unified framework both the Bakery Algorithm and its bounded version, has led to the identification of a new property of mutual exclusion protocols: The existence of a doorway-definable linear order, $\Rightarrow$, which decides the order of the critical sections accesses. In other words, the doorways (and actually the clock events) already establish the priority according to which the processes enter their critical sections, and the remaining parts of the protocols (the wait parts) only ensure that the entries proceed according to this priority. This "Bakery Definability" property is shared by all of the protocols presented here and the protocol of Afek et al. [3]. It may be interesting to find a protocol that does not resemble the Bakery Protocol and yet has the Bakery Definability Property.

There are published solutions to the critical section problem that even have the FCFS property and use much smaller registers than our bounded version protocol (with its 3^n+1 values per register). For example, Lycklama and Hadzilacos [13] only needs registers with 2^4 values. Is there such a solution that satisfy the Bakery Definability property with a fixed number or with a polynomial number $a+n^b$ of values per register (a, and b do not depend on the number of processes n)? in L. Czaja editor, *Proceedings of the CS& P'93 workshop.* pp.7—40, Warszawa 1994.

References

[1] U. Abraham and S. Ben-David and M. Magidor. On Global-Time and Inter-process Communication. In Kwiatkowska, Shields, and Thomas, editors, *Semantics for Concurrency.* Leicester 1990, Proceedings of the International BCS-FACS Workshop, Workshops in Computing series, pp. 311 - 323, Springer-Verlag.

[2] U. Abraham. *Models for Concurrency.* Gordon and Breach, 1999.

[3] Y. Afek, D. Dolev, E. Gafni, M. Merritt and N. Shavit. A bounded First-In, First-Enabled solution to the ℓ-Exclusion Problem. *ACM Transactions on Programming Languages and Systems,* Vol. 16, No. 3, pp. 939-953, 1994.

[4] F. Anger. *On Lamport's interprocess communication. ACM Transactions on Programming Languages and Systems*, Vol. 11 No. 3, pp. 404 - 417, 1989.

[5] S. Ben-David. The global-time assumption and semantics for distributed systems. *7th ACM Symp. on Principles of Distributed Computing*, pp. 223 - 232, 1988.

[6] D. Dolev, and N. Shavit. Bounded concurrent time-stamp systems are constructible. In *Proceedings of the 21 Annual ACM Symposium on Theory of Computing*, Seattle, Washington, pp. 454-465. ACM SIGACT, 1989. (Also *Siam J. Comput.* Vol. 26, 418-455, 1997.)

[7] P. C. Fishburn. *Interval orders and interval graphs.* Wiley, New York (Wiley-Interscience series in discrete mathematics), 1986.

[8] A. Israeli and M. Li. Bounded time-stamps. *Distributed Computing,* 6, Vol.4, pp. 205-209, 1993.

[9] L. Lamport. A new solution of Dijkstra concurrent programming problem in *Communication of the ACM* Vol. 17, 8 pp.453-455, 1974.

[10] L. Lamport. A new approach to proving the correctness of multiprocess programs. *ACM Trans. on Prog. Lang. and systems* 1, pp.84-97, 1979.

[11] L. Lamport. On Interprocess Communication, Part I: Basic formalism, Part II: Algorithms. *Distributed Computing,* Vol. 1, pp. 77 - 101, 1986.

[12] L. Lamport. The mutual exclusion problem: Part I—a theory of interprocess communication; Part II: Statements and solutions. *J. ACM,* 33, pp.313-326, 327-348, 1986.

[13] E. A. Lycklama and V. Hadzilacos. A First-Come-First-Served Mutual-Exclusion Algorithm with Small Communication Variables. *ACM Trans. on Prog. Lang. and Sys.* Vol. 13, pp. 558-576, 1991.

[14] N. Wiener. A contribution to the theory of relative position. *Proc. Camb. Philos. Soc.,* Vol. 17, pp.441-449, 1914.