Theoretical Computer Science ••• (••••) •••–•••

Contents lists available at ScienceDirect

Theoretical Computer Science

www.elsevier.com/locate/tcs

Mutual exclusion as a matter of priority

Yoram Moses *,[1], Katia Patkin

ARTICLE INFO

Article history:
Received 30 December 2015
Received in revised form 5 August 2016
Accepted 15 December 2016
Available online xxxx

Keywords:
Mutual exclusion
Bakery algorithm
Asynchronous computing
Safe registers

ABSTRACT

A new approach to the study and analysis of Mutual Exclusion (ME) algorithms is presented, based on identifying the priority relation that the ME algorithm constructs. It is argued that by analyzing how a process detects that it has priority over all other processes, ME algorithms can be better understood and improved. The approach is illustrated by applying it to Lamport's celebrated Bakery algorithm in the safe register SWMR model. By analyzing how Bakery established and detects priority, cases in which the Bakery algorithm causes processes to block unnecessarily are identified. Namely, a process that already knows that it has priority over another process is made to perform reads and wait on registers of the other process. An optimized version of the Bakery algorithm, called Boulangerie, is proposed, and is shown to be free of any unnecessary blocking. A second contribution of the approach is obtaining a clear explanation for how the Bakery algorithm uses reads from safe registers to detect that a process has priority. Our analysis provides more insight into the workings of the Bakery algorithm than is obtained by other proofs of its correctness.

1. Introduction

Mutual Exclusion (ME) is a fundamental problem in distributed system. Indeed, many consider the introduction of ME by Dijkstra in [5] as the starting point of the field of distributed computing. Intuitively, at the heart of every mutual exclusion algorithm lies a priority relation among processes. To be in its critical section, a process must have priority over all other processes, who are denied access. We suggest that explicitly identifying the priority relation underlying a given ME algorithm is helpful for gaining a better understanding of the workings of the algorithm. This paper applies this approach to Lamport's Bakery algorithm in the safe register SWMR model, providing new insights into the algorithm, and explaining how it obtains its goals despite the use of safe registers. As a result, we identify inefficiencies in the Bakery algorithm and offer optimizations that strictly improve it.

Lamport's Bakery algorithm for mutual exclusion from 1974 [10] is a very influential early solution to mutual exclusion. Lamport states on his website that "*For a couple of years after my discovery of the Bakery algorithm, everything I learned about concurrency came from studying it*" [9], and devoted a significant part of his Turing Lecture at PODC 2014 to mutual exclusion and the Bakery algorithm.[2] The original algorithm is correct under the weak memory assumption that registers are *safe*. A read operation on a safe register that overlaps a write to the same register can return an arbitrary value. As a result, a read performed on such a register may return a value that has never been written to the register.

* Corresponding author.
E-mail address: moses@ee.technion.ac.il (Y. Moses).
[1] Yoram Moses is the Israel Pollak academic chair at the Technion.
[2] For a video of the talk, see http://amturing.acm.org/vp/lamport_1205376.cfm.

http://dx.doi.org/10.1016/j.tcs.2016.12.015
0304-3975/

Please cite this article in press as: Y. Moses, K. Patkin, Mutual exclusion as a matter of priority, Theoret. Comput. Sci. (2016), http://dx.doi.org/10.1016/j.tcs.2016.12.015

There are several proofs of correctness of other variants of the Bakery algorithm [1,4,22,23], as well as mechanical proofs of Lamport's original algorithm [8,16]. Typically, such a proof is based on an inductive invariant, involving a conjunction of several claims, that are all shown to be maintained by every step of the algorithm. While the invariants capture essential aspects of the algorithm, they do not necessarily provide a clear explanation of the algorithm's rationale. Our goal is to provide a new analysis that explains the algorithm in a more transparent fashion.

In a mutual exclusion algorithm, if some process enters its critical section (CS), then, until it leaves the CS, no other process can enter. In this sense, a process in the CS has priority over all others. Every solution to mutual exclusion must construct and depend on such a priority relation, and identifying the priority relation underlying an ME algorithm allows insight into the workings of the algorithm. The current paper studies the priority relation underlying the Bakery algorithm. This provides insight into the workings of the algorithm, and the role of its different components. Moreover, it points to unnecessary blocking and waiting in the Bakery algorithm, and allows a strict improvement to be obtained. Indeed, we prove that the improved version, which we call the *Boulangerie* algorithm, does not suffer from unnecessary blocking.

The Bakery algorithm gets its name from the scheme used in some bakeries or shops, whereby a customer obtains a number upon entry, and the one with the smallest number has priority over the others. In the algorithm, for two processes that are both far enough along in pursuing the critical section (having "entered the bakery"), the one with the smaller number has priority over the other, with ties broken according to the process IDs. For processes that are already in the bakery, priority induces a total ordering. But the Bakery algorithm allows a process to enter the CS even if some or all of the others are not in the bakery. In this case, the asynchrony and concurrency of process operations complicate the picture. Interestingly, the general priority relation implemented by the Bakery algorithm is not even a partial order on processes. Rather, it is an antisymmetric binary relation, which suffices for mutual exclusion. To enter (or be in) the CS, a process must have priority over all others, and antisymmetry ensures that only one process can satisfy this property at any given time.

By studying the Bakery algorithm's priority relation, we find unexpected inefficiencies: In some cases, the algorithm unnecessarily blocks a process from making progress and from entering the CS and requires it to wait unnecessarily for reads of other processes even after the process can deduce that it has priority over them. Two improvements to the algorithm are shown, one of which takes advantage of the fact that a safe register will not show inconsistent readings for a register that is not being written to, and another that removes unnecessary reads and unnecessary blocking when contention for the CS is low. It is shown that the resulting Boulangerie algorithm does not suffer from unnecessary waiting or unnecessary blocking. Deployment of the Boulangerie algorithm can be incremental, in the following sense. Even if an arbitrary subset of the processes follow Boulangerie, while the others follow the original Bakery algorithm, the result is a correct ME algorithm. The Boulangerie users may gain efficiency, but the Bakery users do not suffer any inefficiency or degradation from the fact that others are following Boulangerie. While our analysis provides a rigorous mathematical argument for the correctness of the Boulangerie algorithm, Lamport has performed a mechanical proof of Boulangerie, based on his earlier mechanical proof of the Bakery algorithm [15].

The paper is organized as follows. The next section presents the model of computation and some preliminary definitions. Section 3 reviews the Bakery algorithm and discusses two ways of partitioning it into blocks. It identifies the priority relation underlying the Bakery algorithm, and uses priority to reason about the algorithm and determines the role of central steps of the algorithm. Section 4 identifies superfluous blocking in the Bakery algorithm, describes the Boulangerie algorithm, which is an optimization of the Bakery algorithm, and shows that it does not contain superfluous blocking. In section 5, priority relations and optimizations of other variants of the Bakery algorithm are briefly considered. Finally, section 6 provides some concluding remarks.

2. Preliminary definitions

This paper studies the Bakery algorithm in the asynchronous shared-memory model with safe single-writer, multi-reader (SWMR) registers. Access to these registers is obtained only via **`read`**(*local_var* ← shared_reg) operations, which read a shared register shared_reg into a local variable *local_var*, and by **`write`**(shared_reg ← *x*) operations, which write a (local variable or constant) value *x* to shared_reg. Other than that, computation makes use of local variables only. Thus, for example, to test the value of a shared register (for an **if** statement, for example), a process will read the register into a local variable, and then perform the test. In the SWMR case, every shared register has an "owner" who is the process that can write to it. The others can only perform read operations on the variable. In contrast to local commands such as assignments to local variables, reads and writes are not executed instantaneously. Every **`read`** or **`write`** operation is associated with a *starting* time, which is when the process initiates the operation, and a *completion* time, after which the process proceeds to the next line of code in its program. In between, the process is suspended. (Since registers are SWMR, at most one write to a given register can be active at any given time, and there is no need to specify what happens when writes overlap.)

We assume that shared registers are "safe" in the sense of [13]. Such a register is considered to have a *stable* value at time *t* if its owner is not in the midst of a **`write`** operation to the register at that time. In this case, we define its value at time *t* to be the last value written to it by its owner (or at initialization if no write was performed since then). When a register's value is unstable, we find it convenient to define its value to be '**?**'. While safe registers are sometimes modelled as having a different value at every moment [8,16], we find the direct approach of modelling it as "arbitrary", denoted by '**?**', to be more natural.

Please cite this article in press as: Y. Moses, K. Patkin, Mutual exclusion as a matter of priority, Theoret. Comput. Sci. (2016), http://dx.doi.org/10.1016/j.tcs.2016.12.015

We consider an asynchronous but fair model of computation, in which every process is activated infinitely often, and every **read** and **write** operation that is initiated completes in finite time. At any given time, each process is at a well-defined control state captured by its program counter. At any point in time, the scheduler chooses an arbitrary subset of the processes and "activates" them, causing each of them to take a single step of computation according to its program. A process that is about to perform a **read** or **write** will initiate this operation and move to an "i/o-*suspended*" state. Activating a process in an i/o-suspended state advances it to the next command. Moreover, for a **read**(*local_var* ← shared_reg) it also assigns a value to *local_var*. If the register's value is '**?**', then the scheduler may assign an arbitrary value to *local_var*,[3] and otherwise *local_var* will be assigned the current value of shared_reg.

A ***run*** r of a given program is identified with an infinite sequence of *configurations*. Each configuration determines the values of all local variables, shared registers, as well as the program counters for all processes. (Recall that shared registers may have a value of '**?**' as described above.) In addition, we refer to the state of the computation in the run r at time t by the pair (r,t), which we call a *p*oint.

Mutual exclusion We consider Dijkstra's mutual exclusion problem [5] for $N > 1$ processes in an asynchronous shared-memory setting with safe SWMR registers. The processes are named $1,\ldots,N$. For the problem definition, we follow the exposition in the highly recommended [2]. The code of each process i has a well-defined *critical section* denoted CS_i. Each process also has a well-defined *noncritical section* in which it can stay for an indefinite period (possibly forever). Moreover, a mutual exclusion algorithm must have the property that a process in its noncritical section does not need to participate in the algorithm. In addition to the critical and noncritical sections, every process has two sections of code called the *entry* and *exit* handling the activities leading a process to enter and then to exit the critical section. It is assumed that the code in the critical and in the noncritical sections does not perform write operations on registers that are used in the entry and exit regions. When a process leaves the noncritical section it moves to the entry section, and when it leaves the critical section it moves to the exit section. The model is asynchronous but fair, so that processes not in the noncritical will be scheduled to move infinitely often. In particular, every process that enters its critical section is guaranteed to exit it eventually.

A mutual exclusion algorithm must, at a minimum, satisfy two properties:

- **Exclusion:** At most one process executes its critical section at any time,

and

- **Livelock-freedom:** If some process is in its entry region, then some process will eventually enter its critical section.

Reasoning with propositions It will be convenient to reason about what is true or false in a given configuration. We shall write $(r,t) \models \varphi$ to state that a formula φ is true, or **holds**, at (r,t). Our formulas are boolean combinations[4] of basic propositions, where the basic propositions are either statements regarding the values of variables or registers, or propositions of the form $\mathsf{in}_i(\ell)$, where i is a process and ℓ is a line in the program. We define $(r,t) \models \mathsf{in}_i(\ell)$ to hold if i is either about to execute line ℓ at (r,t), or if the command on line ℓ is a **read** or **write** operation, and i is suspended at (r,t) in the middle of the operation at line ℓ. Given a region L consisting of a set of lines of i's program, it is natural to use $\mathsf{in}_i(L)$ as shorthand for $\bigvee_{\ell\in L} \mathsf{in}_i(\ell)$. (In our analysis, L will be a region of the Bakery algorithm, such as the doorway or the bakery.)

Temporal logic operators Since and Unless We will make use of two binary temporal logic operators, one called *Since* and denoted by '$\mathcal{S}$', and the other called *Unless* (or *weak Until*) and denoted by '$\mathcal{U}$' [18]. They are defined as follows:

- $(r,t) \models \varphi \mathcal{S} \psi$ if for some time $t' \le t$ both
  (a) $(r,t') \models \varphi \wedge \psi$ and
  (b) $(r,m) \models \varphi$ holds for all m in the range $t' \le m \le t$.
  In particular, if $(r,t) \models \varphi \mathcal{S} \psi$ then $(r,t) \models \varphi$.
- $(r,t) \models \varphi \mathcal{U} \psi$ if either
  (a) $(r,t') \models \varphi$ for all times $t' \ge t$, or
  (b) For some time $t' \ge t$ both
    (i) $(r,t') \models \psi$ and
    (ii) $(r,m) \models \varphi$ holds for all m in the range $t \le m \le t'$.
  In particular, if $(r,t) \models \varphi \mathcal{U} \psi$ then $(r,t) \models \varphi$.

[3] E.g., if the value of shared_reg was '0', and '1' is currently being written to it, then a **read** can see an arbitrary value, such as 7456. It is required that the value assigned conform to the register's type, however.

[4] We will freely use boolean operators '¬' (NOT), '∧' (AND) and '⇒' (IMPLIES), with their standard interpretation.

Please cite this article in press as: Y. Moses, K. Patkin, Mutual exclusion as a matter of priority, Theoret. Comput. Sci. (2016), http://dx.doi.org/10.1016/j.tcs.2016.12.015

The Bakery Algorithm for process i.

```
0  Initialize: number[i] = 0;  choosing[i] = false;
1  while true do
2      noncritical section;
3      choosing[i] ← true;
4      number[i] ← 1 + max{number[1], ..., number[N]};
5      choosing[i] ← false;
6      forall the j ≤ N s.t. j ≠ i do
7          await choosing[j] = false ;
8          await number[j] = 0 ∨ ⟨number[i], i⟩ <_L ⟨number[j], j⟩ ;
9      critical section;
10     number[i] ← 0;
```

Fig. 1. The Bakery Algorithm.

3. The Bakery algorithm

Lamport's Bakery algorithm for this model [10] is a protocol $P = (P_1, \ldots, P_N)$, where for each $i = 1, \ldots, N$ the protocol P_i for process i is given in Fig. 1.

Intuitively, every process begins operation in the noncritical section. It proceeds to choose a number in lines **3**–**5**. At this stage, the process checks that it is secure w.r.t. each of the other processes on **6**–**8**, after which it enters its critical section. Upon exiting the critical section, the process resets its number register to 0. On line **8**, we use $<_L$ to denote the lexicographical ordering on pairs. Namely, $\langle \mathsf{number}[i], i\rangle <_L \langle \mathsf{number}[j], j\rangle$ will hold if either $\mathsf{number}[i] < \mathsf{number}[j]$ or $\mathsf{number}[i] = \mathsf{number}[j]$ and $i < j$. For thorough discussions of the Bakery algorithm, see [2,3,17,20]. Traditionally, four main regions are distinguished in the Bakery algorithm: line **2** is called the *noncritical*, lines **3**–**5** are the *doorway*, lines **6**–**9** are the *bakery* and line **10** is the *exit*. Within the bakery region, lines **6**–**8** are called the *testing* region, and line **9** is the *critical section* (CS_i). Our analysis will use a slightly different partition of the algorithm into phases, defined in section 3.2, in which the doorway and bakery regions are modified.

3.1. The Bakery algorithm in the SWMR model

The exposition of the Bakery Algorithm in Fig. 1 above is very close to the original version from [10]. However, our SWMR shared-memory model restricts access to shared registers to consist only of **`read`** and **`write`** operations. Clearly, lines **3**, **5** and **10** correspond to simple **`write`** operations. Lines **4** and **6**–**8** are shorthand for longer bits of code (see, e.g., lines **4.1**–**4.3** in the related algorithm in the Appendix for the more detailed code for line **4**). Observe that the only part of the algorithm in which a process may be blocked waiting for another process to move is in the testing region of lines **6**–**8**. Everywhere else, processes make effective progress whenever they are scheduled to move (i.e., in a wait-free fashion). Finally, we remark that the '**forall** $j < N$ **do**' command on line **6** can be interpreted in two ways: either sequentially (as a standard '**for** $j = \ldots$ **do**' loop, or as a parallel execution of the loop bodies. Our treatment throughout the paper applies equally to either interpretation. For definiteness and ease of exposition, we will assume the sequential interpretation of **forall**.

3.2. Analysis of the Bakery algorithm

A new partition into regions Recall that in a contiguous region of i's code that contains no writes to a particular safe register, this register has a fixed value. Any **`read`** performed to it by other processes while process i is in that region will correctly return this value. Therefore, to facilitate the analysis of the algorithm, we modify the traditional partition slightly. Our purpose is to ensure that the doorway involves no writes to $\mathsf{choosing}[i]$ and the bakery region has no writes to $\mathsf{number}[i]$. To this end, we shrink the doorway region to consist only of line **4**, and shift the bakery region up by one line. We combine the writes on lines **3** and **10** with the noncritical section of line **2** to form a new region that we call the *outside* region. While we abuse the language slightly and maintain the old names for the bakery and doorway regions, we will add a dot on top of the letters used to denote each of the regions, to signal that our regions are slightly modified. The regions under the new partition are given in Table 1.

Observe that the computation of each process i cycles through the sequence of regions $(\mathsf{O}_i; \dot{\mathsf{D}}_i; \dot{\mathsf{B}}_i;)^*$. Starting at the outside, it can move to the doorway and proceed to the bakery. There, after setting $\mathsf{choosing}[i]$ to **false** on line **5**, it proceeds to the testing region, and can only exit the bakery if it passes through its critical section. Upon leaving the critical section, a process is on the outside.

Recall that a configuration of the algorithm records the program counters of the processes. Thus, each process is associated with a unique line ℓ of the program, which the process is either executing, or is about to execute, in that configuration.

Please cite this article in press as: Y. Moses, K. Patkin, Mutual exclusion as a matter of priority, Theoret. Comput. Sci. (2016), http://dx.doi.org/10.1016/j.tcs.2016.12.015

Table 1
A modified partition of the code of Bakery algorithm into regions.

Main region	Symbol	Lines	Subregion	Lines
Outside	O_i	**1–3**, **10**	noncritical	**2**
Doorway	$\dot{\mathsf{D}}_i$	**4**		
Bakery	$\dot{\mathsf{B}}_i$	**5–9**	testing	**6–8**
			critical section, CS_i	**9**

In this case we say that the process is *on* line ℓ. We shall abuse notation slightly and denote the propositions $\mathsf{in}_i(\dot{\mathsf{B}}_i)$, $\mathsf{in}_i(\dot{\mathsf{D}}_i)$ and $\mathsf{in}_i(\mathsf{O}_i)$ by $\dot{\mathsf{B}}_i$, $\dot{\mathsf{D}}_i$ and O_i, respectively. At any given point (r, t) in a run r of the Bakery algorithm, exactly one of $\dot{\mathsf{B}}_i$, $\dot{\mathsf{D}}_i$ or O_i will hold.

According to our new partition of the Bakery algorithm, the choosing[i] register is not written in the (modified) doorway region $\dot{\mathsf{D}}_i$ and number[i] is not written in the (modified) bakery region $\dot{\mathsf{B}}_i$. This implies the following invariants:

Lemma 3.1. *Throughout the Bakery algorithm,*

(a) $\dot{\mathsf{D}}_i \Rightarrow (\mathsf{choosing}[i] = \mathbf{true})$, *and*
(b) $\dot{\mathsf{B}}_i \Rightarrow (0 < \mathsf{number}[i])$, $\dot{\mathsf{B}}_i \Rightarrow (\mathsf{number}[i] \neq \text{`}\mathbf{?}\text{'})$, *and* number[$i$] *remains unchanged while* $\dot{\mathsf{B}}_i$ *holds.*

Proof. Both (a) and (b) are immediate from the partition defined in Table 1 and the sequential nature of the algorithm:

(a) On line **3** process i writes **true** to choosing[i]. It is in the doorway (i.e., $\dot{\mathsf{D}}_i$ holds) only in line **4**, during which it does not perform a write to choosing[i]. Therefore, choosing[i] = **true** in the $\dot{\mathsf{D}}_i$ region, as claimed.
(b) Process i enters the bakery from the doorway region. Upon exiting the doorway (on line **4**), it writes a value greater than 0 to number[i]. Since number[i] is a safe register, we have by definition that the value of number[i] remains unchanged as long as no **write** operation is applied to the register. Since i does not write to number[i] while in the bakery, it follows that number[i] has a positive (and thus non-'**?**') value, and it remains unchanged for as long as i is in the bakery. □

3.3. A priority relation for the Bakery algorithm

Intuitively, we'd like to think of process i as having priority over j at a given point, if it is guaranteed there that j cannot enter the CS before i has entered the CS. It is natural to consider a process i that writes a value to number[i] in line **4** as obtaining a "*ticket*" for entering the CS, consisting of the pair $\langle \mathsf{number}[i], i\rangle$. Tickets are ordered by the lexicographical ordering '$<_{\mathbf{L}}$' on ordered pairs of numbers. Recall that, by Lemma 3.1(b), the value of number[i] (and hence also the ticket $\langle \mathsf{number}[i], i\rangle$) remains unchanged when i is in the bakery region. Since number[h] $\neq$ '**?**' for all processes h in the bakery region, the lexicographical ordering induces a total order on the tickets of all of these processes. Suppose that i and j that are both in the bakery region, and that $\langle \mathsf{number}[i], i\rangle <_{\mathbf{L}} \langle \mathsf{number}[j], j\rangle$. Then i should be able to successfully test against j on line **8**, while j will not be able to do so against i, before i enters the CS. Roughly speaking then, i should be viewed as having priority over j. It is not sufficient to define priority among processes that are both in the bakery region, since a process should be able to enter the CS when other processes are not in the bakery region. To this end, the Bakery algorithm is designed in such a way that if process i enters the bakery before j enters the doorway, then $\langle \mathsf{number}[i], i\rangle <_{\mathbf{L}} \langle \mathsf{number}[j], j\rangle$ will hold if and when j might later join i in the bakery region. The algorithm thus guarantees that j cannot enter the CS before i once $\dot{\mathsf{B}}_i \wedge \mathsf{O}_j$ holds. Indeed, the same is true even if j then advances into the doorway. Based on these observations, we proceed as follows.

Our priority relation for the Bakery algorithm will be based on the "Since" property $\dot{\mathsf{B}}_i \mathcal{S} \mathsf{O}_j$, which in words states that i is now in its bakery region, and it has been in the bakery region ever since (a point in time at which) j was in the outside region. In particular, $\dot{\mathsf{B}}_i \wedge \mathsf{O}_j$ was true at the earlier time. The definition of $\mathcal{S}$ immediately implies that if $\dot{\mathsf{B}}_i \mathcal{S} \mathsf{O}_j$ holds, then $\dot{\mathsf{B}}_i \mathcal{S} \mathsf{O}_j$ continues to hold until $\dot{\mathsf{B}}_i$ ceases to hold (i.e., as long as i remains in the bakery region). This is formally stated by

Lemma 3.2. *Throughout the Bakery algorithm,* $\dot{\mathsf{B}}_i \mathcal{S} \mathsf{O}_j \Rightarrow (\dot{\mathsf{B}}_i \mathcal{S} \mathsf{O}_j) \mathcal{U} \neg \dot{\mathsf{B}}_i$.

We are now ready to define a binary priority relation '$\triangleleft$' among processes for the Bakery algorithm:

Definition 3.1 (PRIORITY). We say that i ***has priority over*** j at (r, t), denoted by $(r, t) \models i \triangleleft j$, if either

(i) $(r, t) \models \dot{\mathsf{B}}_i \wedge \dot{\mathsf{B}}_j \wedge \langle \mathsf{number}[i], i\rangle <_{\mathbf{L}} \langle \mathsf{number}[j], j\rangle$, or
(ii) $(r, t) \models \dot{\mathsf{B}}_i \wedge \neg \dot{\mathsf{B}}_j \wedge \dot{\mathsf{B}}_i \mathcal{S} \mathsf{O}_j$.

Please cite this article in press as: Y. Moses, K. Patkin, Mutual exclusion as a matter of priority, Theoret. Comput. Sci. (2016), http://dx.doi.org/10.1016/j.tcs.2016.12.015

Notice that $i \lhd j$ can hold only when process i is in the bakery (i.e., when $\dot{\mathsf{B}}_i$ holds). When $i \lhd j$ is obtained by Definition 3.1(i), both processes are in the bakery. By Lemma 3.1(b), it follows that both $\mathsf{number}[i] \neq$ **'?'** and $\mathsf{number}[j] \neq$ **'?'**. Therefore, the lexicographical ordering is well-defined in this case. Formally speaking, the priority relation $i \lhd j$ is simply shorthand for a simple temporal formula, obtained by taking the OR of the formulas in parts (i) and (ii) of Definition 3.1.

We note that Lamport has argued in [12] that it is hard to formally specify what one means by a priority relation in a distributed setting. We do not attempt to specify priority. Rather, we show that '$\lhd$' satisfies three properties that, in our opinion, justify our use of the term *priority* for this relation. First, we will show that the relation is antisymmetric, so that if $i \lhd j$ holds then $j \lhd i$ does not hold.[5]

Then, we will show that once $i \lhd j$ holds it will remain true until process i (enters and) exits the critical section. Finally, we will show that a process can enter or be in its critical section only if it has priority over all other processes. We start with the first property:

Lemma 3.3. *The priority relation '$\lhd$' is antisymmetric.*

Proof. We need to show that $(r,t) \not\models (i \lhd j) \wedge (j \lhd i)$ for all points (r,t) that arise in the Bakery algorithm. Assume that $(r,t) \models (i \lhd j) \wedge (j \lhd i)$, by way of contradiction. Then, by definition, we have that $(r,t) \models \dot{\mathsf{B}}_i \wedge \dot{\mathsf{B}}_j$. By Definition 3.1, both $\langle \mathsf{number}[j], j\rangle <_{\mathbf{L}} \langle \mathsf{number}[i], i\rangle$ and $\langle \mathsf{number}[i], i\rangle <_{\mathbf{L}} \langle \mathsf{number}[j], j\rangle$ hold at (r,t). This is a contradiction, since '$<_{\mathbf{L}}$' is an ordering relation. □

Our next goal is to show that once the priority relation $i \lhd j$ holds, it persists for as long as i is in the bakery. We first show that if $i \lhd j$ holds by Definition 3.1(ii), i.e., when j is not in the bakery, then $i \lhd j$ will continue to hold even if j enters the bakery (at which time $i \lhd j$ will hold by clause (i)), for as long as i remains in the bakery.

Lemma 3.4. *The formula* $\dot{\mathsf{B}}_i \mathcal{S} \mathsf{O}_j \Rightarrow i \lhd j$ *is true throughout the Bakery algorithm.*

Proof. By Definition 3.1(ii), $\left(\neg \dot{\mathsf{B}}_j \wedge \dot{\mathsf{B}}_i \mathcal{S} \mathsf{O}_j\right) \Rightarrow i \lhd j$ is valid. We will show that $\left(\dot{\mathsf{B}}_j \wedge \dot{\mathsf{B}}_i \mathcal{S} \mathsf{O}_j\right) \Rightarrow i \lhd j$ is valid as well. Let r be a run of the bakery algorithm, and assume that $(r,t) \models \dot{\mathsf{B}}_j \wedge \dot{\mathsf{B}}_i \mathcal{S} \mathsf{O}_j$. It follows that $(r,t) \models \dot{\mathsf{B}}_i \wedge \dot{\mathsf{B}}_j$ and for some time $t_1 < t$ both (a) $(r,t_1) \models \dot{\mathsf{B}}_i \wedge \mathsf{O}_j$, and (b) for all times m in the range $t_1 \leq m \leq t$ we have that $(r,m) \models \dot{\mathsf{B}}_i$. Without loss of generality assume that t_1 is the latest time with this property. By Lemma 3.1(b) we have that $\mathsf{number}[i] \neq$ **'?'** and it remains unchanged throughout the interval $[t_1, t]$. Between time t_1 at which j is in O_j and time t at which j is in the bakery, process j enters the doorway on line **4**, after which $\mathsf{number}[j] > \mathsf{number}[i]$ holds as long as both processes remain in the bakery region. It follows that $(r,t) \models i \lhd j$ by Definition 3.1(i). □

Lemma 3.4 can be used to show that if $i \lhd j$ holds, then it remains true as long as $\dot{\mathsf{B}}_i$ holds. Formally:

Corollary 3.5. *Throughout the Bakery algorithm,* $i \lhd j \;\Rightarrow\; (i \lhd j) \mathcal{U} \neg \dot{\mathsf{B}}_i$.

Proof. The claim is equivalent to $\dot{\mathsf{B}}_i \mathcal{S} (i \lhd j) \;\Rightarrow\; i \lhd j$. We will prove the latter. Thus, assuming that $(r,t) \models \dot{\mathsf{B}}_i \mathcal{S} (i \lhd j)$, we will show that $(r,t) \models i \lhd j$. By definition of $\mathcal{S}$ we have that $(r,t) \models \dot{\mathsf{B}}_i$, and for some time $t' \leq t$ both (i) $(r,t') \models \dot{\mathsf{B}}_i \wedge (i \lhd j)$ and (ii) $(r,m) \models \dot{\mathsf{B}}_i$ holds for all times m in the range $t' \leq m \leq t$. We prove the claim by induction on $k = t - t'$. The claim is immediate if $t - t' = 0$ since then $t = t'$ and $(r,t') \models \dot{\mathsf{B}}_i \wedge (i \lhd j)$. Let $t - t' = k > 0$, and assume inductively that the claim is true for $k-1$. Since $(t-1) - t' = k - 1 \geq 0$, we have by the inductive assumption that $(r, t-1) \models i \lhd j$. By definition of the '$\lhd$' relation, we consider two cases:

(a) $(r, t-1) \models \dot{\mathsf{B}}_j \wedge \langle \mathsf{number}[i], i\rangle <_{\mathbf{L}} \langle \mathsf{number}[j], j\rangle$. In this case, if $(r,t) \models \dot{\mathsf{B}}_j$ then the values of $\mathsf{number}[i]$ and of $\mathsf{number}[j]$ are unchanged from $(r, t-1)$, so that $(r,t) \models \dot{\mathsf{B}}_i \wedge \dot{\mathsf{B}}_j \wedge \langle \mathsf{number}[i], i\rangle <_{\mathbf{L}} \langle \mathsf{number}[j], j\rangle$ and $(r,t) \models i \lhd j$ holds by Definition 3.1(i). If, however, $(r,t) \not\models \dot{\mathsf{B}}_j$ then j moved out of the bakery region $\dot{\mathsf{B}}_j$ between time $t-1$ and t, and so $(r,t) \models \mathsf{O}_j$. It follows that $(r,t) \models \dot{\mathsf{B}}_i \wedge \neg \dot{\mathsf{B}}_j \wedge \dot{\mathsf{B}}_i \mathcal{S} \mathsf{O}_j$, and so $(r,t) \models i \lhd j$ holds by Definition 3.1(ii).

(b) $(r, t-1) \models \dot{\mathsf{B}}_i \mathcal{S} \mathsf{O}_j$. Since $(r,t) \models \dot{\mathsf{B}}_i$ we have that $(r,t) \models \dot{\mathsf{B}}_i \mathcal{S} \mathsf{O}_j$, and so $(r,t) \models i \lhd j$ follows by Lemma 3.4. □

Suppose that $(r,t) \models i \lhd j$. Then Corollary 3.5 ensures that $i \lhd j$ will continue to be true in r from time t as long as i does not exit the bakery region $\dot{\mathsf{B}}_i$. By definition, $i \lhd j$ implies that i is in the bakery region. Since the only way it can

[5] The relation '$\lhd$' is not a partial order, because it is not transitive: It is possible for $(r,t) \models i \lhd j$ to hold due to clause (ii) of Definition 3.1 and for $(r,t) \models j \lhd k$ to hold by clause (i), while $(r,t) \not\models i \lhd k$ because $(r,t) \models \dot{\mathsf{B}}_i \wedge \dot{\mathsf{D}}_k \wedge \neg \dot{\mathsf{B}}_i \mathcal{S} \mathsf{O}_k$. We remark that the transitive closure of '$\lhd$' is a partial order, but the Bakery algorithm detects priority only using the basic clauses in the definition of $\lhd$.

Please cite this article in press as: Y. Moses, K. Patkin, Mutual exclusion as a matter of priority, Theoret. Comput. Sci. (2016), http://dx.doi.org/10.1016/j.tcs.2016.12.015

exit the bakery region is by exiting the critical section, Corollary 3.5 ensures in particular that j will not enter CS_j until i (perhaps enters and then) exits CS_i.

3.4. Proving mutual exclusion

Intuitively, the iteration of the testing region by process i (lines **6** to **8**) performed with parameter j is intended to establish and detect i's priority over j. For ease of exposition, we will denote by $\mathbf{7}_j$ and $\mathbf{8}_j$ the instances of lines **7** and **8** performed by i in this iteration. We now turn to seeing how this is achieved. First, we consider the precise role that the wait for $\mathsf{choosing}[j] = \mathbf{false}$ on line $\mathbf{7}_j$ serves. We show that if this wait by i succeeds, it is guaranteed that j has been out of the doorway region at some point during the wait. Formally, we state this as:

Lemma 3.6. $\dot{\mathsf{B}}_i \,\mathcal{S} \neg \dot{\mathsf{D}}_j$ *holds whenever process* i *leaves line* $\mathbf{7}_j$.

Proof. Recall that $\mathsf{choosing}[j] = \mathbf{true}$ when process j enters $\dot{\mathsf{D}}_j$. Moreover, in the $\dot{\mathsf{D}}_j$ region, process j does not perform writes to $\mathsf{choosing}[j]$. Thus, a read of $\mathsf{choosing}[j]$ that completely overlaps $\dot{\mathsf{D}}_j$ will necessarily return **true**. Suppose that process i leaves line $\mathbf{7}_j$ at (r,t), having successfully completed the wait on line $\mathbf{7}_j$. Its last r/w operation on line $\mathbf{7}_j$ is a **read** of the register $\mathsf{choosing}[j]$, which returns **false**. Since $\mathsf{choosing}[j] = \mathbf{true} \neq$ **'?'** whenever j is in the doorway, it follows that at some time $t' < t$ this **read** operation did not overlap $\dot{\mathsf{D}}_j$. Moreover, at all times between t' and t in r, process i is in the bakery. It follows that $(r,t) \models \dot{\mathsf{B}}_i \,\mathcal{S} \neg \dot{\mathsf{D}}_j$, as claimed. □

Lemma 3.6 implies the following very useful fact:

Corollary 3.7. *If* i *completes* $\mathbf{7}_j$ *at* (r,t) *and* $(r,t) \not\models i \lhd j$, *then* $(r,t) \models \dot{\mathsf{B}}_i \wedge \dot{\mathsf{B}}_j \wedge \langle \mathsf{number}[j], j\rangle <_{\mathbf{L}} \langle \mathsf{number}[i], i\rangle$.

Proof. Suppose that i completes $\mathbf{7}_j$ at (r,t). By Lemma 3.6 we have that $(r,t) \models \dot{\mathsf{B}}_i \,\mathcal{S} \neg \dot{\mathsf{D}}_j$. Thus, there is an earlier time $t' < t$ such that $\dot{\mathsf{B}}_i$ holds continuously between t' and t, and $(r,t') \models \neg \dot{\mathsf{D}}_j$. If $(r,t') \models \mathsf{O}_j$, then $(r,t) \models \dot{\mathsf{B}}_i \,\mathcal{S}\, \mathsf{O}_j$ and so $(r,t) \models i \lhd j$ holds by Lemma 3.4. The assumption that $(r,t) \not\models i \lhd j$ implies that $(r,t) \models \neg\big(\dot{\mathsf{B}}_i \,\mathcal{S}\, \mathsf{O}_j\big)$, a contradiction. Hence, $(r,t') \models \neg \mathsf{O}_j$, and since $(r,t') \models \neg \dot{\mathsf{D}}_j$ we have that $(r,t') \models \dot{\mathsf{B}}_j$. Since (a) $\dot{\mathsf{B}}_i$ holds continuously between times t' and t, (b) $(r,t) \models \neg\big(\dot{\mathsf{B}}_i \,\mathcal{S}\, \mathsf{O}_j\big)$, and (c) $(r,t') \models \dot{\mathsf{B}}_j$, it follows that $\dot{\mathsf{B}}_i \wedge \dot{\mathsf{B}}_j$ also holds continuously between times t' and t in r. In particular, $(r,t) \models \dot{\mathsf{B}}_i \wedge \dot{\mathsf{B}}_j$. Recall that we have that $(r,t) \not\models i \lhd j$ holds by assumption, and we thus obtain that $(r,t) \models \dot{\mathsf{B}}_i \wedge \dot{\mathsf{B}}_j \wedge \langle \mathsf{number}[j], j\rangle <_{\mathbf{L}} \langle \mathsf{number}[i], i\rangle$, as claimed. □

When $\dot{\mathsf{B}}_i \wedge \dot{\mathsf{B}}_j \wedge \langle \mathsf{number}[j], j\rangle <_{\mathbf{L}} \langle \mathsf{number}[i], i\rangle$ holds, we have in particular that $\mathsf{number}[j]$ is stable for as long as $\dot{\mathsf{B}}_j$ holds. If i later reads a larger value for $\mathsf{number}[j]$ (so that the test for $\langle \mathsf{number}[i], i\rangle <_{\mathbf{L}} \langle \mathsf{number}[j], j\rangle$ succeeds), this will indicate that j has left the bakery region. But then $\dot{\mathsf{B}}_i \mathcal{S} \mathsf{O}_j$ is true—i was in the bakery since j was outside. By Lemma 3.4, we have $i \lhd j$ at that point. Line $\mathbf{8}_j$ waits precisely until such a value is read by i. Conequently, we can now show

Lemma 3.8. $i \lhd j$ *holds whenever process* i *leaves line* $\mathbf{8}_j$.

Proof. Suppose that process i leaves line $\mathbf{8}_j$ at (r,t), having successfully completed the wait. Let $t' < t$ be the most recent time at which i completed $\mathbf{7}_j$. If $(r,t') \models i \lhd j$ then $(r,t) \models i \lhd j$ holds by Corollary 3.5, since i does not leave the bakery between time t' and time t. Otherwise, $(r,t') \models \dot{\mathsf{B}}_i \wedge \dot{\mathsf{B}}_j \wedge \langle \mathsf{number}[j], j\rangle <_{\mathbf{L}} \langle \mathsf{number}[i], i\rangle$ holds, by Corollary 3.7. But line $\mathbf{8}_j$ is completed after reading a value k for $\mathsf{number}[j]$ that must be larger than the one for which $\langle \mathsf{number}[j], j\rangle <_{\mathbf{L}} \langle \mathsf{number}[i], i\rangle$ held at time t'. It follows that j must have left the bakery region at some time t'' in the range $t' < t'' \leq t$. At time t'' we have that $(r,t'') \models \dot{\mathsf{B}}_i \mathcal{S} \mathsf{O}_j$, implying that $(r,t) \models \dot{\mathsf{B}}_i \mathcal{S} \mathsf{O}_j$ as well, and so $(r,t) \models i \lhd j$, as claimed. □

We can now show that a process i can enter or be in the CS (i.e., be in line **9**) only if $i \lhd j$ holds ***for all*** $j \neq i$. Formally:

Theorem 3.9. *Throughout the Bakery algorithm,* $\mathsf{CS}_i \;\Rightarrow\; \bigwedge_{j \neq i} i \lhd j$.

Sketch of proof. Recall that the testing region of the algorithm at lines **6** to **8** is fully contained in the bakery region. Hence, by Corollary 3.5, if $i \lhd j$ holds in that region, it remains true as long as $\dot{\mathsf{B}}_i$ holds. By the time process i reaches line **9** it has completed line $\mathbf{8}_j$ for all $j \neq i$. For each $j \neq i$, Lemma 3.8 implies that $i \lhd j$ holds at some point when i is in the testing region. The claim follows. □

Please cite this article in press as: Y. Moses, K. Patkin, Mutual exclusion as a matter of priority, Theoret. Comput. Sci. (2016), http://dx.doi.org/10.1016/j.tcs.2016.12.015

By the antisymmetry of the '$\lhd$' relation (Lemma 3.3), at most one process can have priority over all others, and so Theorem 3.9 immediately yields:

Corollary 3.10. *The Bakery algorithm guarantees mutual exclusion: At most one process is in the CS at any time.*

3.5. Liveness and fairness of the Bakery algorithm

In addition to the mutual exclusion property, the priority relation and our modified partition facilitate reasoning about other properties of the Bakery algorithm. For example, the algorithm is known to satisfy a form of FCFS (first-come first-serve) fairness. In [2,17,22], for example, the FCFS property shown is that if i enters (our) bakery region before j enters the doorway, which in our terminology means that $\dot{\mathsf{B}}_i \mathcal{S} \mathsf{O}_j$ holds, then i will enter its critical section before j does. This follows immediately from Lemma 3.4, Corollary 3.5, Theorem 3.9. We can now state and prove slightly finer fairness properties of the Bakery algorithm:

Theorem 3.11 *(Fairness). In all runs r of the Bakery algorithm and times t:*

(a) *If $(r,t) \models \dot{\mathsf{B}}_i$ then no process j can enter the CS twice after time t in r before i enters the CS at least once, and*
(b) *If $(r,t) \models \dot{\mathsf{D}}_i$ then no process j can enter the CS three times after time t in r before i enters the CS at least once.*

Sketch of proof. For part (a), suppose that i is in the bakery region while j is in its critical section, at some time $t' \geq t$. When j exits the critical section $\dot{\mathsf{B}}_i \mathcal{S} \mathsf{O}_j$ will hold, and so $i \lhd j$ will hold by Lemma 3.4. By Corollary 3.5 this priority will persist as long as i is in the bakery region. It follows that process j will not be able to enter its critical section again before i leaves the bakery region, having visited the critical section. The claim follows.

For part (b), suppose that i is in the doorway and j is in its critical section. Process j will not be able to enter its critical section again without successfully completing line $\mathbf{7}_i$, which can happen only after i leaves the doorway and enters the bakery region. By part (a), process j will not be able to enter its critical section again (i.e., for the third time) before i visits its critical section. Here again, the claim follows. □

The progress assumptions of section 3.1 imply that any process that leaves the noncritical section (line **3**) will reach the bakery in a finite amount of time, in a wait-free fashion. For completeness, we use this fact to state and prove a natural liveness condition for the Bakery algorithm as follows (similar proofs appear elsewhere; see, e.g., [3]):

Theorem 3.12. *If at least one process reaches the bakery region, then at least one process will enter the CS.*

Sketch of proof. Let $k > 0$ be the number of processes in the bakery region at (r,t). We prove the claim by induction on $N-k$. The base case is $k = N$, in which all processes are in the bakery region. In particular, they all have stable and well-defined number values. So let i_0 be the process with the '$<_{\mathrm{L}}$'-minimal value. Thus, $i_0 \lhd j$ holds for all $j \neq i_0$ and no process can enter the CS before i_0 does. We claim that i_0 will complete the testing region successfully and enter the CS. Since all processes are in the bakery, they will all move beyond line **5** in finite time. At that point, all choosing$[j]$ values are **false**, and so the success for i_0 of the test on line $\mathbf{7}_j$ is guaranteed. Clearly, since no process in the bakery writes to its number register, i_0 is able to successfully read each of these, and succeeds in completing the wait of line $\mathbf{8}_j$, for each j.

For the inductive step, suppose that $0 < k < N$ and assume inductively that the claim holds for $k+1$. Since '$<_{\mathrm{L}}$' is a total order, there is a minimal index i_0 among the processes that are in the bakery at (r,t). If all other $N-k$ processes are in the noncritical section throughout the time at which i_0 performs the testing region of the bakery region, then process i_0 will complete lines **7** and **8** for all processes and will enter the CS. (In particular, for the ones in the noncritical it will read choosing$[j]$ = **false** and number$[j] = 0$.) Otherwise some process j_0 must be in, or enter, the doorway while i_0 performs the testing region. At some later time j_0 will also enter the bakery. If some process has exited the bakery by the time j_0 enters the bakery, this process has entered the CS, as required. Otherwise, there will be $k+1$ processes in the bakery and the claim follows by the inductive hypothesis. □

Since, by assumption, a process that leaves the noncritical section is guaranteed to reach the bakery region in finite time, Theorem 3.12 immediately implies:

Corollary 3.13. *The Bakery algorithm satisfies Livelock-freedom.*

4. *Boulangerie*: a better Bakery algorithm

In the testing region of the Bakery algorithm (lines **6**–**8**) process i detects that $i \lhd j$ holds, for each of the $j \neq i$. As long as i is unable to establish that $i \lhd j$ based on the checks in lines $\mathbf{7}_j$ and $\mathbf{8}_j$, process i will block waiting for j to

Please cite this article in press as: Y. Moses, K. Patkin, Mutual exclusion as a matter of priority, Theoret. Comput. Sci. (2016), http://dx.doi.org/10.1016/j.tcs.2016.12.015

make progress. Our analysis showed how succeeding in the tests on both lines guarantees that $i \triangleleft j$ holds. We now wish to consider whether the blocking imposed by $\mathbf{7}_j$ and $\mathbf{8}_j$ is always justified.

Optimizing for low contention Let us first consider the blocking imposed by line $\mathbf{7}_j$. Roughly speaking, the justification for this blocking is that j may be in the doorway region, and when it exits the doorway and enters the bakery region it might have a small number, and thus a ticket with better priority than i's ticket. However, suppose that the testing process i has obtained $\mathsf{number}[i] = 1$. Then the only processes j that can *ever* have a better ticket are ones whose ID is smaller than i (so that $\langle 1, j\rangle <_{\mathrm{L}} \langle 1, i\rangle$). It follows that when $\mathsf{number}[i] = 1$, there is no need to perform $\mathbf{7}_j$ and $\mathbf{8}_j$ for values $j > i$. To avoid this form of unnecessary blocking, we can replace line **6** of the Bakery algorithm by the following two lines:

6a **if** $\mathsf{number}[i] = 1$ **then** $\mathsf{Limit} \leftarrow i - 1$ **else** $\mathsf{Limit} \leftarrow N$;

6b **forall the** $j \le \mathsf{Limit}$ **s.t.** $j \neq i$ **do**

This optimization affects the behavior of the algorithm only in cases in which processes exit the doorway with a number value of 1. However, even though the values of $\mathsf{number}[i]$ can grow without bound in the Bakery algorithm, we claim that the case of $\mathsf{number}[i] = 1$ is not always a boundary case. Observe that such unbounded growth requires continuous contention for the critical section. Critical sections come in many flavors, and in many cases they do not experience continuous contention. Indeed, it is generally believed that contention for a critical section is rare in a well-designed system (see, e.g., [14]). Notice that whenever all processes are in the noncritical region at once, even for a brief instant, their number values are all 0. The next process to leave the doorway will do so with a number value of 1 (others may attain a number value of 1 as well). Therefore, when mutual exclusion is applied to a critical section that repeatedly experiences low contention, this optimization will repeatedly result in a reduction in the amount of blocking.[6]

Taking advantage of inconsistent reads We now consider the blocking imposed by line $\mathbf{8}_j$. As our analysis shows (in Corollary 3.7), when process i completes $\mathbf{7}_j$, either $i \triangleleft j$ is already true, or both i and j are in the bakery, and j has a better ticket. Very roughly speaking, process i blocks on $\mathbf{8}_j$ until it reads a value that contradicts the fact that j has a better ticket. Suppose, for example, that $\mathsf{number}[i] = 10$ and that i reads a value of 5 for $\mathsf{number}[j]$. It blocks correctly, since 5 could be the stable value of $\mathsf{number}[j]$. Observe that $\mathsf{number}[j]$ is a safe register, and so **`read`** operations on it may return arbitrary, and inconsistent, values when $\mathsf{number}[j] =$ '**?**'. So now suppose that i performs another **`read`** on $\mathsf{number}[j]$, and obtains a value, say 4 or 6, that is different from 5. This still corresponds to a better ticket for j than $\langle 10, i\rangle$. But there is a subtle point here. As long as j is in the bakery it performs no writes on $\mathsf{number}[j]$. Thus, $\mathsf{number}[j]$ is stable and all reads to it must return the same value. If i reads two different values for $\mathsf{number}[j]$ while blocking on line $\mathbf{8}_j$, it has proof that j was on the outside at least during one of these reads. Since $\dot{\mathsf{B}}_i$ holds at that point, it follows that $i \triangleleft j$ is true, and i can stop blocking on j and move on to test the next process.

We can avoid this case of unnecessary blocking as follows. When $\mathsf{number}[j]$ is read for the second consecutive time or later on line $\mathbf{8}_j$, let $\mathtt{previous}(\mathsf{number}[j])$ denote the previous value read by i from $\mathsf{number}[j]$, or let it be undefined if no such read has yet occurred. (See lines $\mathbf{8}_j\mathbf{a}$ to $\mathbf{8}_j\mathbf{e}$ in Fig. 2.) Then we can replace line $\mathbf{8}_j$ in the Bakery algorithm by the following:

$\mathbf{8}_j$ **await** $\mathsf{number}[j] = 0 \ \vee\ \langle \mathsf{number}[i], i\rangle <_{\mathrm{L}} \langle \mathsf{number}[j], j\rangle \ \vee\ \mathsf{number}[j] \neq \mathtt{previous}(\mathsf{number}[j])$;

We call the optimized variant of the Bakery algorithm that incorporates both changes the ***Boulangerie*** algorithm. Its full detailed description is given in Fig. 2. Its partition is analogous to that of the Bakery algorithm: Lines **4.1**–**4.3** constitute the doorway region $\dot{\mathsf{D}}_i$, while lines **5**–**9** are $\dot{\mathsf{B}}_i$.

The optimization for the case of $\mathsf{number}[i] = 1$, which is beneficial under low contention, utilizes an aspect of priority that the Bakery algorithm admits, but does not try to detect. Namely, if $\mathsf{number}[i] = 1$ and $i < j$, then j will not be able to beat i to the CS. Since Boulangerie makes explicit use of this fact, we need to modify '$\triangleleft$' slightly in order to capture the notion of priority that corresponds to the Boulangerie algorithm:

Definition 4.1. $i \blacktriangleleft j$ iff $(i \triangleleft j) \vee \left(\dot{\mathsf{B}}_i \wedge \neg \dot{\mathsf{B}}_j \wedge \mathsf{number}[i] = 1 \wedge i < j\right)$.

By Definition 4.1, the new priority relation '$\blacktriangleleft$' is strictly stronger than '$\triangleleft$'. When both processes are in the bakery region, the two relations coincide and reduce to lexicographic ordering. A very similar analysis as that used to establish Lemma 3.3 and Corollary 3.5 can show (for the proof see section 4.1):

Lemma 4.1. *In the Boulangerie algorithm, both*

(a) *The priority relation '$\blacktriangleleft$' is antisymmetric, and*

(b) *throughout the Boulangerie algorithm,* $i \blacktriangleleft j \Rightarrow \left(i \blacktriangleleft j\right) \mathcal{U} \neg \dot{\mathsf{B}}_i$.

[6] A particular case in which the savings with this optimization can be striking is when there are $N = 2$ processes. In this case, process 1 will not need to perform the testing region when $\mathsf{number}[1] = 1$.

Please cite this article in press as: Y. Moses, K. Patkin, Mutual exclusion as a matter of priority, Theoret. Comput. Sci. (2016), http://dx.doi.org/10.1016/j.tcs.2016.12.015

Detailed Boulangerie Algorithm for process i

```
0   Initialize: num[i] = number[i] = 0;  choosing[i] = tmp_c = false;  Limit = N;
    prev_n = tmp_n = ⊥;
1   while true do
2       noncritical section;
3       write(choosing[i] ← true) ;
4.1     forall the j ≤ N; j ≠ i do read(num[j] ← number[j]) ;
4.2     num[i] ← 1 + max{num[1], ..., num[N]};
4.3     write(number[i] ← num[i]) ;
5       write(choosing[i] ← false) ;
6a      if number[i] = 1 then Limit ← i − 1 else Limit ← N;
6b      forall the j ≤ Limit s.t.  j ≠ i do
7_j         repeat read(tmp_c ← choosing[j]) until (tmp_c = false);
8_j a       tmp_n ← ⊥;
8_j b       repeat
8_j c           prev_n ← tmp_n;
8_j d           read(tmp_n ← number[j])
8_j e       until (tmp_n = 0 ∨ ⟨num[i], i⟩ <_L ⟨tmp_n, j⟩ ∨
                          (tmp_n ≠ prev_n ∧ prev_n ≠ ⊥));
9       critical section;
10.1    num[i] ← 0;
10.2    write(number[i] ← num[i]) ;
```

Fig. 2. The Boulangerie Algorithm.

In fact, Lemma 4.1 is true for the Bakery algorithm itself, as well as for any protocol $P_{\mathsf{mix}} = (P_1, \ldots, P_N)$ in which some of the P_i's are Bakery and some are Boulangerie. As a result, all such protocols P_{mix} are correct mutual exclusion protocols, and this implies that the optimization of the Bakery algorithm proposed by Boulangerie can be implemented incrementally. We will now show that Boulangerie solves mutual exclusion.

4.1. Analysis of the Boulangerie algorithm

In this section we will present the safety proof for our optimized algorithm, using the new priority relation '$\blacktriangleleft$'. The proof is analogous to the proof for the Bakery algorithm presented in sections 3.3 and 3.4. We make some modifications to account for the new algorithm and the new relation. As in the earlier case, we will first show that the priority relation is antisymmetric, so that if $i \blacktriangleleft j$ holds then $j \blacktriangleleft i$ does not hold. Then, we will show that once $i \blacktriangleleft j$ was established it will remain true as long as process i is in the bakery region. Finally, we will show that a process can enter or be in its critical section only if it has priority over all other processes.

Proof of Lemma 4.1(a). The proof is identical to the proof for Lemma 3.3, which was conducted by contradiction. Since the third clause in the definition of '$\blacktriangleleft$' applies when $\dot{\mathsf{B}}_i \wedge \neg \dot{\mathsf{B}}_j$, only the first clause of the definition of '$\triangleleft$', Definition 3.1(i), applies under the assumption (by contradiction) that both $i \blacktriangleleft j$ and $j \blacktriangleleft i$ hold. The proof is thus the same as it is for the relation '$\triangleleft$'. □

As stated earlier, we now aim to show that once the priority relation $i \blacktriangleleft j$ holds, it persists as long as i is in the bakery. For the clauses that coincide with those of the definition of '$\triangleleft$', the proof is identical for '$\blacktriangleleft$'. It remains to show that if $i \blacktriangleleft j$ holds by the third clause of '$\blacktriangleleft$', i.e., when j is not in the bakery, and $\mathsf{number}[i] = 1$, then $i \blacktriangleleft j$ will continue to hold as long as i remains in the bakery, even if j enters the bakery.

Lemma 4.2. *The formula* $\dot{\mathsf{B}}_i \mathcal{S} \mathsf{O}_j \Rightarrow i \blacktriangleleft j$ *is true throughout the Boulangerie algorithm.*

Please cite this article in press as: Y. Moses, K. Patkin, Mutual exclusion as a matter of priority, Theoret. Comput. Sci. (2016), http://dx.doi.org/10.1016/j.tcs.2016.12.015

Proof. The proof is completely analogous to that of Lemma 3.4. □

Lemma 4.3. *The formula* $\left(\dot{\mathsf{B}}_i \wedge \mathsf{number}[i] = 1 \wedge i < j\right) \Rightarrow i \blacktriangleleft j$ *is true throughout the Boulangerie algorithm.*

Proof. By Definition 4.1, $\left(\neg \dot{\mathsf{B}}_j \wedge \left(\dot{\mathsf{B}}_i \wedge \mathsf{number}[i] = 1 \wedge i < j\right)\right) \Rightarrow i \blacktriangleleft j$ is valid. It remains to show that $i \blacktriangleleft j$ is true also if $\dot{\mathsf{B}}_j \wedge \left(\dot{\mathsf{B}}_i \wedge \mathsf{number}[i] = 1 \wedge i < j\right)$. In this case we have that $\dot{\mathsf{B}}_i \wedge \dot{\mathsf{B}}_j$, and so $i \blacktriangleleft j$ is equivalent to $\langle \mathsf{number}[i], i\rangle <_{\mathrm{L}} \langle \mathsf{number}[j], j\rangle$. Since $\mathsf{number}[j] \geq 1$ and $i < j$, we have that $\langle 1, i\rangle <_{\mathrm{L}} \langle \mathsf{number}[j], j\rangle$, and the claim follows by clause (i). □

We can now complete the proof of Lemma 4.1:

Proof of Lemma 4.1(b). The proof is similar to the proof of Corollary 3.5. Suppose that $(r,t) \models \dot{\mathsf{B}}_i \, \mathcal{S} \, (i \blacktriangleleft j)$. By definition of $\mathcal{S}$ we have that $(r,t) \models \dot{\mathsf{B}}_i$. So like in Corollary 3.5, we need to show that $(r,t) \models i \blacktriangleleft j$. By definition of $\mathcal{S}$ we have that there exists a time $t' \leq t$ such that $(r,t') \models \dot{\mathsf{B}}_i \wedge i \blacktriangleleft j$. We assume for all times m in the range $t' \leq m \leq t$ we have that $(r,m) \models \dot{\mathsf{B}}_i$. If $(r,t') \models i \blacktriangleleft j$ holds due to $(r,t') \models i \lhd j$, then the proof is identical to Corollary 3.5. Using the inductive hypothesis in the proof of Corollary 3.5, we only need to show that the claim holds for the third clause, that is, in the case that $(r,t-1) \models \neg \dot{\mathsf{B}}_j \wedge \mathsf{number}[i] = 1 \wedge i < j$. The value of $\mathsf{number}[i]$ is unchanged from $(r,t-1)$ by Lemma 3.1. We have that $(r,t) \models \dot{\mathsf{B}}_i \wedge \mathsf{number} = 1 \wedge i < j$, and so $(r,t) \models i \blacktriangleleft j$ follows by Lemma 4.3. □

As in section 3.4, the jth iteration of the testing region by process i is intended to establish and detect i's priority over j. Line **7** serves the same role in Boulangerie as in the Bakery algorithm. Thus, if the wait by i in line $\mathbf{7}_j$ succeeds, it is guaranteed that j has been out of the doorway region at some point during the wait, as is formally stated in Lemma 4.4. We have seen that often when process i leaves line $\mathbf{7}_j$ it already has priority over j, in the rest of the cases j has priority over i, as captured by Corollary 4.5, and the role of the second wait in the testing region is simply to ensure that, if i did not have priority over j, then priority has since been gained. This is stated in Lemma 4.6.

Lemma 4.4. $\dot{\mathsf{B}}_i \, \mathcal{S} \neg \dot{\mathsf{D}}_j$ *holds whenever process* i *leaves line* $\mathbf{7}_j$.

Lemma 4.4 implies the following very useful fact:

Corollary 4.5. *If* i *completes* $\mathbf{7}_j$ *at* (r,t) *and* $(r,t) \not\models i \blacktriangleleft j$, *then* $(r,t) \models \dot{\mathsf{B}}_i \wedge \dot{\mathsf{B}}_j \wedge \langle \mathsf{number}[j], j\rangle <_{\mathrm{L}} \langle \mathsf{number}[i], i\rangle$.

The proofs of Lemma 4.4 and Corollary 4.5 are identical to those of Lemma 3.6 and Corollary 3.7. If $\mathsf{number}[i] \neq 1$ then $\mathsf{Limit} = N$, so Bakery and Boulangerie are the same for process i, up until (and including) $\mathbf{7}_j$. Otherwise, if $\mathsf{number}[i] = 1$ then $\mathsf{Limit} = i - 1$, so process i completes line $\mathbf{7}_j$ only for $j \leq i - 1$.

Lemma 4.6. $i \blacktriangleleft j$ *holds whenever process* i *leaves line* $\mathbf{8}_j$.

The proof of Lemma 4.6 is similar to the proof of Lemma 3.8. If $\mathsf{number}[i] \neq 1$ then $\mathsf{Limit} = N$, so process i completes line $\mathbf{8}_j$ for all processes in the system just as in the Bakery protocol. Otherwise, if $\mathsf{number}[i] = 1$ then $\mathsf{Limit} = i - 1$, so process i completes line $\mathbf{8}_j$ only for $j \leq i - 1$. Notice that the only difference in line $\mathbf{8}_j$ in the Boulangerie protocol versus the Bakery protocol is the addition of the following disjunction: the previous read of $(\mathsf{number}[j])$ is not equal to the latest read of $\mathsf{number}[j]$. This addition captures exactly the point in the proof of Lemma 3.8, where j has left the bakery region. If two consecutive reads lead to different values, then we can conclude that during sometime between these reads j was not in the bakery region, otherwise its shared variables should be stable, implying $\dot{\mathsf{B}}_i \mathcal{S} \mathsf{O}_j$, hence $i \blacktriangleleft j$.

Finally, we show that a process can enter or be in its critical section only if it has priority over all other processes.

Theorem 4.7. *Throughout the Boulangerie algorithm,* $\mathsf{CS}_i \Rightarrow \bigwedge_{j \neq i} i \blacktriangleleft j$.

Sketch of proof. If $\mathsf{number}[i] \neq 1$ then $\mathsf{Limit} = N$ or if $\mathsf{number}[i] = 1$ then $\mathsf{Limit} = i - 1$, for $j \leq i - 1$, the proof follows from that of Theorem 3.9. We need to show that if $\mathsf{number}[i] = 1$ then $i \blacktriangleleft j$ is also true for all $j > i$. Let r be a run of Boulangerie, and assume that for some time $t' < t$, it holds that $(r,t'-1) \models \neg \dot{\mathsf{B}}_i$, $(r,t') \models \dot{\mathsf{B}}_i \wedge \mathsf{number}[i] = 1$ and both (a) process i at line **9** and (b) for all times m in the range $t_1 \leq m \leq t$ we have that $(r,m) \models \dot{\mathsf{B}}_i$. Without loss of generality assume that t' is the latest time with these properties. By Lemma 4.3 it follows that $(r,t') \models i \blacktriangleleft j$ for all $j > i$ and by Lemma 4.1(b), if $i \blacktriangleleft j$ holds, it remains true as long as $\dot{\mathsf{B}}_i$ holds. The claim follows. □

Please cite this article in press as: Y. Moses, K. Patkin, Mutual exclusion as a matter of priority, Theoret. Comput. Sci. (2016), http://dx.doi.org/10.1016/j.tcs.2016.12.015

By the antisymmetry of the '◄' relation (Lemma 4.1(a)), at most one process can have priority over all others, and so Theorem 4.7 immediately yields:

Corollary 4.8. *The Boulangerie algorithm satisfies the Exclusion property.*

4.2. No unnecessary blocking in Boulangerie

In a precise sense, Boulangerie strictly dominates the Bakery algorithm. Moreover, it achieves savings without needing to modify any of the **write**'s performed by the Bakery algorithm. We observed that the Bakery protocol causes processes to wait unnecessarily in some cases, and proposed the Boulangerie protocol as an optimization in which such waiting is avoided. In Theorem 4.9 we show that in the new protocol, there is no unnecessary waiting. We do this by showing that as long as a process has not finished the testing region in Boulangerie, it does not know that the CS is empty. This is a formal claim using the theory of knowledge in distributed systems (see [6,7]). Here we consider knowledge based on the agent's complete local history, to show that the protocol cannot be improved. We do not need to repeat the theory here. Rather, we point out that the claim will be proved if we show that the Boulangerie algorithm satisfies the following property: If a process i has not completed the testing region at a point (r,t), then there is another run r' of Boulangerie and a time t' such that process i has the same local history (by which we mean the complete sequence of events observed so far by process i) at both (r,t) and (r',t'), and the CS is not empty at (r',t'). We prove this by assuming that at some point (r,t) process i is in the testing region and has not yet completed testing with respect to j, and constructing a run r' and identifying a time t' at which j is in CS_j and, in addition, i has the same history at (r,t) and at (r',t'). Formally, we proceed as follows:

Theorem 4.9. *Let (r,t) be a point of the Boulangerie algorithm in which process i is in the testing region, and assume that i has not completed line $\mathbf{8}_j$ since entering the bakery. If $\mathsf{number}[i] > 1$ or $j < i$, then based on the sequence of events it has seen thus far, process i **does not know** that j is out of the critical section at (r,t).*

Proof. We refer to the detailed version of the Boulangerie algorithm presented in Fig. 2. The case in which there are $N = 2$ processes is simple and left to the reader. We prove the claim for $N \geq 3$. Let us consider a point (r,t) in a run r of the Boulangerie algorithm at which process i is in the bakery region but has not completed line $\mathbf{8}_j$, and $j < i$ or $\mathsf{number}[i] > 1$. We shall construct another run r' and identify a time t' such that $(r',t') \models \mathsf{CS}_j$, and process i's local history up to time t' in r' is the same as it history up to time t in run r.

The theorem's assumptions are made w.r.t. the point (r,t). If $(r,t) \models \mathsf{CS}_j$ then the claim trivially holds: Defining $(r',t') \triangleq (r,t)$ we have that the CS in nonempty at (r',t') since $(r',t') \models \mathsf{CS}_j$, and i obviously has the same local history at (r,t) and at (r',t').

It remains to consider the case of $(r,t) \models \neg\mathsf{CS}_j$. By assumption, $(r,t) \models \dot{\mathsf{B}}_i$. Let $t_0 < t$ be the last time in the run r at which i is in the noncritical section. Such a time exists because i starts out in the noncritical section at time 0, and clearly $(r,0) \models \neg\,\dot{\mathsf{B}}_i$. In the rest of the proof all references to regions and line numbers will refer to their occurrences after time t_0. Recall that i is assumed to be in the testing region at time t. Its local history since time t_0 consists of the writing of $\mathsf{choosing}[i]$ on lines **3** (and later also **5**), as well as the $num[h]$ values it has read in the doorway. This is followed by the values it reads for the various $\mathsf{choosing}[h]$ and the values $\mathsf{number}[h]$ it reads in the testing region up to time t. Let r' coincide with r up to time t_0. We extend (r',t_0) by activating the processes that are not in the noncritical section in a round-robin manner. In particular, a process in its critical section is activated repeatedly until it exits it. Since we assume that processes must exit the CS after a finite amount of time, we proceed until every process other than i reaches the noncritical section. (A process in the noncritical section is no longer activated at this stage.) An inductive proof as in that of Theorem 3.12 shows that all processes (other than i) will eventually be in the noncritical section at once. Let the time at which this happens in r' be t_1. Notice that i is not activated between times t_0 and t_1, and so i's local history at (r',t_1) is the same as at (r,t_0). Process i's local history at (r,t) is its local history at (r,t_0), followed by the list of **read** results it observes and **write** operations that it performs in r from time t_0 to time t. This includes the $\mathsf{number}[h]$ values that i reads (into local variables $num[h]$) on line **4**, and values it reads on lines **7** and **8**. We need to extend (r',t_1) to a point (r',t') in such a way that i will observe the exact same sequence of reads and writes between (r',t_1) and (r',t'). Let K be the maximal process ID smaller than or equal to j for which i has performed at least one read on line $\mathbf{7}_K$ by time t in r (and $K = 0$ if no such process exists). We now construct the rest of r' by cases.

- Suppose that $\mathsf{number}[i] = 1$ and $j < i$ at (r,t). Starting at (r',t_1) we move both i and j to the doorway, and have them both read all of the registers as 0, and exit the doorway with $\mathsf{number}[i] = \mathsf{number}[j] = 1$.
  - (a) Process i proceeds and enters the bakery region and completes the **write** on line **5** and reaches line **7**, with the same local history as it does in r. Now we have i "simulate" in r' the testing phase in r as follows. For every process $h \leq K$ such that $h \notin \{i,j\}$, we do as follows:
    - Process h advances to line **3** and starts the **write** to $\mathsf{choosing}[h]$ (and then it suspends). Process i performs as many **read** operations of $\mathsf{choosing}[h] =$ **true** in the course of line $\mathbf{7}_h$ in r' as it does in the run r, followed by a **read** of $\mathsf{choosing}[h] =$ **false** if it succeeds in $\mathbf{7}_h$ in r. These are possible because $\mathsf{choosing}[h]$ is a safe register, and the **read** operations on $\mathsf{choosing}[h]$ are performed concurrently with the write by h.

* Process h completes the **write** of choosing[h] $\leftarrow$ **true**, proceeds through the doorway reading the current values (or 0 for registers who are '**?**'), and begins the **write** to number[h] on line **4.3**.
* In case process i started performing the **read** on line $\mathbf{8}_h$ by time t in r, then at this point i moves to line $\mathbf{8}_h$ and performs as many **read** operations as it does at $\mathbf{8}_h$ in the run r, reading the same values in the same order. This is possible, again, because the reads are performed concurrently with the write of number[h] by h on line **4.3**. If i completes $\mathbf{8}_h$ in r by time t, it completes it in r' as well with the same view.
* At this point, h completes the **write** of number[h] (which receives a value > 1) and performs the **write** of choosing[h] $\leftarrow$ **false** on line **5**.

(b) By assumption, i does not complete line $\mathbf{8}_j$ successfully by time t in r. If it performs any part of $\mathbf{7}_j$ and $\mathbf{8}_j$ at all, then $K = j$ and process i now performs the same steps of the testing phase on j as in r:
* process i performs the same number of **read** operations on choosing[j] that result in **true** as it does on $\mathbf{7}_j$ in r, and then (if this happens in r) it reads **false** to proceed to $\mathbf{8}_j$. This is possible because j is writing choosing[j] on line **5**.
* If i performs **read** operations on number[j] in line $\mathbf{8}_j$ in r, then all of them return the same value 1 (mentioned above), because it does not succeed to complete line $\mathbf{8}_j$, by assumption. In r' process i will complete the same number of **read** operations on number[j]. Process i will read the same values as in r, without j needing to write to number[j], since $\langle 1, i\rangle <_{\mathrm{L}} \langle \text{number}[j], j\rangle$ only if number[j] $= 1$, as it is.

(c) At this time the reads on $\mathbf{7}_j$ and $\mathbf{8}_j$ are completed (if they are ever performed) and at this stage in the run r', process i has the same local history as it does in (r, t).

(d) Process j is advanced to lines **6**–**8**. It succeeds in all of its tests, because choosing is **false** for all processes, and it has a lexicographically better value than everyone else ($j < i$ and all $h \notin \{i, j\}$ has number[h] > 1). Process j proceeds to enter the CS. Denote the time at which this happens by t'. Since j's progression through **6**–**8** did not affect i's local history, we have that $(r', t') \models \mathsf{CS}_j$ and i has the same local history as in (r, t), as required.

- Now suppose that number[i] > 1 at (r, t). Recall that all processes are in the noncritical section at (r', t_1). Let c be the value that i obtains in r from reading number[j] on line $\mathbf{8}_j$ in the bakery in r in case $j = K$ and such read occurs. (If multiple reads occur on line $\mathbf{8}_j$, then they are all of the same value, because of the test on line $8_j\mathbf{e}$.) Otherwise, let $c = 1$. In this case, starting at (r', t_1) our goal will be to have i read the same values in the doorway as it does in r between t_0 and t, and have j read $c - 1$, so that $c =$ number[j] $<$ number[i] when they enter the bakery region. We proceed to construct r' from (r', t_1), as follows.

(a) We are assuming that $N > 2$. So we choose some node $h_0 \notin \{i, j\}$ and move it solo through the bakery and the CS, until it reaches line **10** and suspends while writing number[h_0] as 0.

(b) We move all processes $h \notin \{i, h_0\}$ through the doorway, having all of the processes with $h < j$ read all registers as 0, and the ones with $h \geq j$ read all 0's, except that they read number[h_0] as $c - 1$. Each such process h then suspends on line **4.3** while writing number[h].

(c) Process h_0 moves through the noncritical to doorway, and reads 0 for all numbers if $h_0 < j$ and $c-1$ for all numbers except number[i] $= 0$ if $h_0 > j$. It then continues to line **4.3** and suspends while writing the value to number[h_0].

(d) We move process i through the doorway, where it reads the same values into the $num[h]$ variables, for all $h \neq i$, as it does in r between t_0 and t. These values can be read, because all registers are in the course of being written, and hence have value '**?**'. Process i proceeds to write number[i] as in r (to a value larger than 1, by assumption), and enters the bakery region with the same local history as it does in r.

(e) Process j completes the **write** of c to number[j], and proceeds to line **5**, suspending while writing to choosing[j].

(f) All processes $h \notin \{i, j\}$ complete writing number[h] on line **4.3** and complete the **write** of **false** to choosing on line **5**.

(g) At this point, process i completes the **write** on line **5** in r'.

(h) One by one, each of the processes $h < j$ such that $h \neq i$ passes through the testing region and the CS, and continues to reach the noncritical section. It successfully reads all choosing[x] registers as **false** to complete $\mathbf{7}_x$ for all x, and successfully completes the test on line $\mathbf{8}_x$ based on the lexicographical test of $\langle \text{number}[h], h\rangle <_{\mathrm{L}} \langle \text{number}[x], x\rangle$. The test for choosing[$j$] $=$ **false** succeeds because j is suspended while writing to choosing[j] on line **5**.

(i) Now i simulates the testing phase of r in r'. For every $h \leq K$ such that $h \notin \{i, j\}$, in ascending order:
* process h moves to line **3** and suspends while writing choosing[h].
* process i performs the same number of **read** operations on choosing[h] that result in **true** as it does on $\mathbf{7}_h$ in r, and then (if this happens in r) it reads **false** to proceed to $\mathbf{8}_h$.
* process h proceeds through the doorway and suspends while writing to number[h] on **4.3**.
* at this stage, process i performs the same number of **read** operations on number[h] as in r, and succeeds in completing $\mathbf{8}_h$ if it does so in r.

(j) If $K = j$ then i now repeats the same part of the testing phase on j as in r:
* process i performs the same number of **read** operations on choosing[j] that result in **true** as it does on $\mathbf{7}_j$ in r, and then (if this happens in r) it reads **false** to proceed to $\mathbf{8}_j$. This is possible because j is writing choosing[j] on line **5**.

Please cite this article in press as: Y. Moses, K. Patkin, Mutual exclusion as a matter of priority, Theoret. Comput. Sci. (2016), http://dx.doi.org/10.1016/j.tcs.2016.12.015

* If i performs **read** operations on number$[j]$ in line $\mathbf{8}_j$ in r, then all of them return the same value c (mentioned above), because it does not succeed to complete line $\mathbf{8}_j$, by assumption. In r' process i will complete the same number of **read** operations on number$[j]$. Since number$[j] = c$, it will read the same values as in r.

(k) At this stage in the run r', process i has the same local history as it does in (r, t). It remains to show that j can now move into the CS without involving i. First, we move all $h \leq K$ that are still in the doorway into the testing region. They all have number$[h] > c$. Now all processes are either in the noncritical section or in the bakery, choosing$[h] =$ **false** for all processes h, and j has the smallest ticket $\langle$number$[j], j\rangle$. So we move j through the testing region and into the CS. This is the desired point (r', t'), at which i has the same local history as at (r, t), and $(r', t') \models \mathsf{CS}_j$. The proof is thus complete. □

We remark that recent work on the connection between knowledge and action in distributed systems presents a theorem called the *Knowledge of Preconditions Principle*. It implies that, since the critical section must be empty when process i enters CS_i, the process must ***know*** that the critical section is empty when it enters [19]. Theorem 4.9 shows that the Boulangerie algorithm is optimally efficient in this sense: It ensures that a process that is trying to enter its critical section will do so at the first instant at which it knows that the critical section is empty.

5. Proving and improving other variants of the Bakery algorithm

Variants of the Bakery algorithm have been considered for other, related models of distributed systems, such as asynchronous message-passing [21] and asynchronous shared memory with regular or atomic registers [3,11,17,20]. A similar approach can assist in the analysis of those algorithms as well. We now mention without proof some of the connections and implications to these variants. (A detailed analysis is beyond the scope of this paper.)

The Bakery algorithm for regular registers replaces line **3** of the Bakery algorithm by a write of number$[i] \leftarrow c$, where c is some value greater than 0, and removes lines **5** and **7** (see [11]). A regular register is one in which the value read by a **read** that overlaps a **write** is either the new value being written, or the old value before the **write**. A completely analogous proof to the one in section 3.4, using the priority relation '◀' rather than '◁', shows that the algorithm satisfies mutual exclusion as well as fairness and liveness. Regular registers cannot have inconsistent reads as safe registers can. As a result, the optimization of the Bakery algorithm for inconsistent reads does not apply in this case. It is possible to improve the protocol slightly by avoiding testing against processes $j > i$ when number$[i] = 1$. The gain is somewhat smaller, however, because in the Bakery algorithm for regular registers of [11] process i does not block until $j > i$ leaves the doorway in the optimization case, so it performs exactly one unnecessary read for every $j > i$.

The Ricart and Agarwala variant of the Bakery algorithm is a mutual exclusion algorithm for asynchronous message passing. It can be shown to use and detect the '◁' priority relation. In the bakery region, a process i asks each process $j \neq i$ for "permission" to enter the CS. The request is positively acknowledged by j only if $i \lhd j$ holds. This can simplify the statement of invariants for this algorithm. A similar improvement as Boulangerie can be obtained in this case when number$[i] = 1$ (this is a local variable in that algorithm), although it is strictly a boundary case. That is because the "number$[i]$" value in this algorithm is obtained from a (logical or physical) clock. Hence, number$[i] = 1$ can be true at most once for every process, and so the optimization is not significant.

6. Conclusion

This paper offers a behind-the-scenes look at the workings of Lamport's Bakery algorithm in the rather challenging case of safe registers. We identified the priority relation that the algorithm implements and detects, and used it to formally capture the role of the main components of the algorithm. Based on the analysis, we were able to find two ways in which the Bakery algorithm admits unnecessary and potentially costly blocking. An improved version, called Boulangerie, fixes these inefficiencies and is shown to contain no unnecessary blocking.

Reading from safe registers provides limited information about the state of the system, since a value can be read from such a register without ever being written to it. Indeed, when a test whether number$[j] = 0$ succeeds, this does *not* mean that number$[j] = 0$ was in fact true at some point during the **read** operation. As a result, it is tricky to interpret the tests performed on lines $\mathbf{7}_j$ and $\mathbf{8}_j$ of the Bakery algorithm. Indeed, their intuitive interpretation differs from their true behavior.

As Lemma 3.6 illustrates, a value read from a safe register can provide information about the register and about the state of its writer ***by way of elimination***. Namely, if v is read, then no other value $w \neq v$ was the stable value of the register throughout the time of the read operation. With the proper definition of a priority relation among the processes, these insights allow a process to conclude that it has priority over other processes by performing the tests on lines $\mathbf{7}_j$ and $\mathbf{8}_j$ (Lemma 3.8), as discussed in section 4.

Our analysis of the Bakery algorithm is based on the observation that every mutual exclusion algorithm must break symmetry among the processes by implementing a priority relation of some sort. We saw that Lamport's original Bakery algorithm is based on the relation $i \lhd j$, which combines a lexicographic relation over the register values with a temporal condition. In particular, i has priority over j if i is in the bakery ever since j was on the "outside", before j ever entered the doorway. A slightly different relation, '◀', corresponds to the Boulangerie algorithm and to the Bakery algorithm for regular registers of [11].

Please cite this article in press as: Y. Moses, K. Patkin, Mutual exclusion as a matter of priority, Theoret. Comput. Sci. (2016), http://dx.doi.org/10.1016/j.tcs.2016.12.015

This paper proposes that ME algorithms should be studied by considering the priority relations that they construct and detect. This essential aspect of ME, which has been in the background all along, can be brought to the fore. We believe that the view of Mutual Exclusion as a matter of creating and detecting priority proposed in this paper promises to provide new insight into other existing ME algorithms, and may lead to the development of new ones. As our analysis proves in the case of the Bakery algorithm, even very familiar solutions can be seen in a new light, resulting in a better understanding as well as genuine improvements in the algorithm. Lamport reports in [15] that he has been able to machine-verify the Boulangerie algorithm by fairly simple modifications of his machine proof of the Bakery algorithm. We see this as confirming both the quality of his verification tools and the fact that the logic underlying Boulangerie is a refinement and direct improvement of the Bakery algorithm. It is instructive that the Boulangerie algorithm is a conservative extension of the Bakery algorithm. Deploying it in a setting in which processes are already using the Bakery algorithm can safely be performed incrementally, by gradually moving nodes to use the improved Boulangerie algorithm. In a system where some of the nodes follow the original Bakery algorithm and others follow Boulangerie correctly implements Mutual Exclusion. The latter nodes are simply more efficient in some cases.

Acknowledgements

We wish to thank Leslie Lamport for mechanically proving the Boulangerie algorithm, and the first author wishes to thank Fred Schneider for useful discussions on the topic of this paper. Special thanks go to the anonymous referees for an outstanding job. This work was supported in part by ISF grant 1520/11.

References

[1] U. Abraham, Logical classification of distributed algorithms (Bakery Algorithms as an example), Theoret. Comput. Sci. 412 (25) (June 2011) 2724–2745.
[2] J.H. Anderson, Lamport on mutual exclusion: 27 years of planting seeds, in: Proceedings of the 20th ACM PODC Conference, ACM, 2001, pp. 3–12.
[3] H. Attiya, J. Welch, Distributed Computing: Fundamentals, Simulations, and Advanced Topics, vol. 19, John Wiley & Sons, 2004.
[4] K. Chaudhuri, D. Doligez, L. Lamport, S. Merz, Verifying safety properties with the TLA+ proof system, in: Proceedings of the 5th ICAR Conference, Springer-Verlag, 2010, pp. 142–148.
[5] E.W. Dijkstra, Solution of a problem in concurrent programming control, Commun. ACM 8 (9) (Sept. 1965) 569.
[6] R. Fagin, J.Y. Halpern, Y. Moses, M.Y. Vardi, Reasoning About Knowledge, MIT Press, 2003.
[7] J.Y. Halpern, Y. Moses, Knowledge and common knowledge in a distributed environment, J. ACM 37 (3) (1990) 549–587.
[8] W.H. Hesselink, Mechanical verification of Lamport's Bakery Algorithm, Sci. Comput. Program. 78 (9) (2013) 1622–1638.
[9] L. Lamport, My writings, Available at http://research.microsoft.com/en-us/um/people/lamport/pubs/pubs.html.
[10] L. Lamport, A new solution of Dijkstra's concurrent programming problem, Commun. ACM 17 (8) (Aug. 1974) 453–455.
[11] L. Lamport, A new approach to proving the correctness of multiprocess programs, ACM Trans. Program. Lang. Syst. 1 (1) (Jan. 1979) 84–97.
[12] L. Lamport, What it means for a concurrent program to satisfy a specification: why no one has specified priority, in: Proceedings of the 12th ACM POPL Conference, ACM, 1985, pp. 78–83.
[13] L. Lamport, On interprocess communication. Part I: Basic formalism, Distrib. Comput. 1 (2) (1986) 77–85.
[14] L. Lamport, A fast mutual exclusion algorithm, ACM Trans. Comput. Syst. 5 (1) (Jan. 1987) 1–11.
[15] L. Lamport, A TLA+ mechanical proof of the Boulangerie Algorithm, http://research.microsoft.com/en-us/um/people/lamport/tla/boulangerie.html, 2015.
[16] L. Lamport, The TLA+ hyperbook, http://research.microsoft.com/en-us/um/people/lamport/tla/hyperbook.html, 2015.
[17] N.A. Lynch, Distributed Algorithms, Morgan Kaufmann Publishers Inc., 1996.
[18] Z. Manna, A. Pnueli, The Temporal Logic of Reactive and Concurrent Systems, Springer, 1992.
[19] Y. Moses, Relating knowledge and coordinated action: the knowledge of preconditions principle, in: Proceedings of TARK 2015, 2016, arXiv:1606.07525.
[20] M. Raynal, D. Beeson, Algorithms for Mutual Exclusion, MIT Press, 1986.
[21] G. Ricart, A.K. Agrawala, An optimal algorithm for mutual exclusion in computer networks, Commun. ACM 24 (1) (1981) 9–17.
[22] D. Rosenzweig, E. Börger, Y. Gurevich, The Bakery Algorithm: yet another specification and verification, in: E. Börger (Ed.), Specification and Validation Methods, Oxford University Press, Inc., 1995, pp. 231–243.
[23] E. Sedletsky, A. Pnueli, M. Ben-Ari, Formal verification of the Ricart–Agrawala Algorithm, in: Proceedings of the 20th FST TCS Conference, Springer-Verlag, 2000, pp. 325–335.

Please cite this article in press as: Y. Moses, K. Patkin, Mutual exclusion as a matter of priority, Theoret. Comput. Sci. (2016), http://dx.doi.org/10.1016/j.tcs.2016.12.015