Theoretical Computer Science 412 (2011) 2724–2745

Contents lists available at ScienceDirect

Theoretical Computer Science

journal homepage: www.elsevier.com/locate/tcs

Logical classification of distributed algorithms (Bakery algorithms as an example)

Uri Abraham

Departments of Mathematics and Computer Science, Ben-Gurion University of the Negev, Beer-Sheva, Israel

ARTICLE INFO

Article history:
Received 22 April 2009
Received in revised form 22 July 2010
Accepted 22 January 2011
Communicated by A. Avron

Keywords:
Tarski structures
Classification of algorithms
Event structures
Mutual exclusion
Distributed algorithms
Models of concurrency
Many-sorted logics

ABSTRACT

We argue that logical descriptions of distributed algorithms can reveal key features of their high-level properties, and can serve to classify and explicate fundamental similarities even among superficially very dissimilar algorithms. As an illustration, we discuss two distinct mutual-exclusion algorithms: the Bakery algorithm of Lamport is for shared memory, and the Ricart and Agrawala version is for message passing. It is universally agreed that they are both instances of "the Bakery algorithm" family, but is there a formal expression of this affinity? Here we present logical properties expressed naturally in Tarskian event structures that allow us to capture the similarities precisely. We use the notions of low-level and high-level events to organize the comparison. We find a set of properties expressed in quantification language which are satisfied by every Tarskian system execution that models a run by either one of the protocols, and we suggest these properties as a formal explication for the similarity of the two algorithms. An abstract proof shows that these common properties imply the mutual exclusion, and the informal arguments explain the sense in which they capture the essence of the two Bakery algorithms.

1. Introduction

There are many examples of different algorithms that appear to be so similar that it makes sense to view them as variants of each other and to classify them as a family of algorithms. An example that can illustrate this and which is investigated in this paper is the Bakery algorithm family which includes the original algorithm of Lamport (1974) [12], the message-passing variant of Ricart and Agrawala (1981) [23], and a dozen of other variants. We deal with the following question in this paper. How can we formally define such a family of algorithms that appear to be so close on an intuitive level?

The simplest answer would be to classify the algorithms by specifying that which they are designed to achieve. For example, sorting algorithms are designed to order an input array, and the sorting algorithm family is defined as all those algorithms that achieve this aim. Another example for a purpose-based classification is the family of critical section algorithms. That is the family of those algorithms that achieve mutual exclusion (no two critical section executions are concurrent). Additional properties, such as First Come First Served can be used to define a proper subfamily, and with more properties one can define an even narrower family, but we will probably never reach the Bakery algorithm family with this type of specifications, and we claim that a deeper level is required in order to find out what makes two Bakery like algorithms similar.

Our thesis is that it is at the correctness proof level that the similarities lie. That is, we claim that the proof of the mutual-exclusion property for the Bakery algorithm of Lamport and the corresponding proof for the Ricart and Agrawala algorithm have a meaningful part that is exactly the same, and it is this identical part in the correctness proof that formally expresses the similarity between the two algorithms. Moreover, not every correctness proof can highlight these similarities. In our

E-mail address: abraham@cs.bgu.ac.il.

0304-3975/$ – see front matter
doi:10.1016/j.tcs.2011.01.030

experience, correctness proofs that refer to Tarskian event structures (see below) are most suitable for this aim, and so the purpose of this paper is to show with a detailed example that predicate languages and Tarskian structures can be used in order to classify concurrent algorithms that are employed in distributed systems.

We would like to say already at this early stage that we do not view this quest as a formal exercise. When properly done, classification is an important aspect of the scientific investigation of asynchronous algorithms which can deepen our understanding and ability to use them.

Bakery algorithms

The Bakery algorithm is a critical section algorithm for $N \geq 2$ processes, introduced by Lamport in 1974 [12]. It is an intriguing algorithm: simple and short enough to be grasped in a glance, and yet subtle and involving important aspects of concurrency. It is based on an intuitively simple idea which can be described in two or three sentences, and still offers a challenge for any formal correctness proof. It has been extensively studied: one can easily count more than ten publications that describe variants of the Bakery algorithm. In fact, Lamport himself has published four Bakery algorithms in addition to the classical algorithm of [12]: the algorithm in [13], the algorithms in [15], and the algorithm of [16]. There are several published Bakery algorithms that employ only bounded values (e.g. [1,11,24,25]), and there are algorithms that employ the message-passing mode of communication instead of shared memory registers (e.g. [23]). In this article we begin to investigate the following question: In what sense are these diverse algorithms "Bakery algorithms"? We suggest a format or framework in which an answer to such questions can be given, namely by providing some predicate language properties that are common to a family of algorithm. Finding useful, common properties of algorithms that use different modes of communication (shared memory and message passing) is a challenge because such properties reside above the communication level and thus reveal the essence of the algorithms.

Not being a native speaker, it took me a while to realize that the "bakery" in not a place where bread is baked, but rather where it is sold together with other sugared and glazed products. The "algorithm" is not a baker's recipe, but a method to serve the costumers: upon entry a customer takes a number from a dispenser machine (which dispenses its numbers in increasing order), and is then served after those clients with smaller numbers have been served. We note that this practical ticket dispenser algorithm has only a limited value as the intuitive paradigm for Lamport's algorithm, because accesses to the dispenser need some mutual-exclusion mechanism whereas the Bakery algorithm does not rely on any assumed mutual exclusion, and its registers need not even be atomic.

On the intuitive and very informal level, the Bakery algorithm and its variants are characterized by their having two stages: in the first stage a process P "takes a ticket" and obtains some evidence about its position in the service ordering, and in the second stage P waits until all other processes acknowledge that they have no objection that P is served. In the terminology of Raynal [22], these are permission based mutual-exclusion algorithms. Our mission here is to formalize this intuitive idea, and to find a set of properties (axioms as we call them) that reflect it and which are shared by the following two seemingly quite different Bakery algorithms:

1. The Take-A-Number algorithm (in Section 3) is a simple variant of the original Bakery algorithm of Lamport [12] in which the processes get their ticket numbers from an assumed "dispenser" rather than reading the registers of all other processes.
2. The Ricart and Agrawala algorithm [23] (in Section 6) which uses messages for communication.

Although these algorithms are obviously different, the first is designed for the shared memory mode of communication and the other for the message-passing mode, it is universally agreed that they are in the same family.[1] One may be tempted to think that there is a simple reason for this resemblance. There are many algorithms that implement shared memory communication within the message-passing mode. That is, algorithms that employ messages for communication and "mimic" the behavior of shared registers read and write operations (see the Simulation part in [3]). So one can take the shared memory Bakery algorithm and replace each read and write operation by a call to the mimicking algorithm. The resulting algorithm would certainly be a message-passing copy of the Bakery algorithm, and if the Ricart and Agrawala algorithm were of this type, then our quest for formally expressing the similarity would be rather trivial. But the Ricart and Agrawala and the other message-passing "Bakery like" algorithms are not obtained by a simple substitution of this type, and a deeper answer has to be found.

We deal here with a variant of Lamport's Bakery algorithm: in the original algorithm each process finds its timestamp not by some dispenser machine but by reading the timestamps of all other processes and choosing a larger timestamp for itself. In a sense, this self-sufficient algorithm is nicer as it does not rely on any machine. Nevertheless for my aim of exemplifying the usage of Tarskian structures to highlight the similarities between two different algorithms, the simplest possible examples that are still nontrivial are required, and the variant algorithm's simplicity is an advantage.

Road map of the paper and discussion of related works

This is a rather long paper with a necessary coverage of background material. In Section 2 Tarskian system executions are defined, and in Section 5 the message-passing mode of communication and the causal ordering [14] are defined. Although

[1] Ricart and Agrawala themselves mention this resemblance in their paper [23].

this is not new material, it is presented here from an unusual point of view—one which employs event-based Tarskian structures (predicate language interpretations) and hence these two long introductory sections are indispensable. Yet the main point of the paper is accessible even to those readers who would prefer the following shortcuts to the essential parts of the paper.

1. Read Section 3. This section describes our simplified shared memory Bakery algorithm (named Take-A-Number) and proves its mutual-exclusion property.
2. Then in Section 4 this mutual-exclusion proof is reconsidered at an abstract level which is not directly related to any specific algorithm. This short section is the heart of the paper. We prove there that the mutual-exclusion property is a consequence of a list of high-level properties.
3. The Ricart and Agrawala algorithm is presented in Section 6. We prove that the abstract high-level properties of Section 4 hold in any execution of the Ricart and Agrawala algorithm. The conclusion is that this algorithm satisfies the mutual-exclusion property, and, more importantly, it satisfies this property for the same reason that the Bakery Take-A-Number does. Hence the affinity between these two protocol is formally expressed.

Related work

System specification is a central issues in computer science. For example, in model checking, the basic question is to "mechanically determine if the system meets a specification expressed in propositional temporal logic" (Clarke [6]). Given a temporal logic formula τ, the family S of all systems that satisfy τ is well defined, and hence temporal logic can serve to classify algorithms by having in one class all algorithms whose executions satisfy τ. Yet specification in the model checking sense is mainly concerned with the automatic verification question, whereas we are interested here in clarifying the intuitive resemblance between algorithms. It seems to be an open problem whether temporal logic can highlight the similarity between the two algorithms presented here.

Besides the very large body of specification literature, it seems that not many publications address this classification question. In fact I am aware of just two such publications. The work of Raynal [22] is motivated by a similar quest of classifying critical section algorithms, but his survey is not intended to give a formal, mathematical definition and it remains at an intuitive level. Then (as explained to me by Yoram Moses) the knowledge-based analysis in [9] (and other papers in this area) is motivated by the desire to consider a family of algorithms from a higher and more abstract point of view that reveals their common knowledge-based properties.

There are however quite a few publications that promote the usage of event structures and their first-order and higher-order logic as we do. Here are three examples of such research directions.

1. Lamport's work on TLA+ is based on the notion of a first-order language interpretation, that is a Tarskian structure (see [19]). Nevertheless, although the notion of Tarskian interpretation appears in the definition of a state, a *behavior* is a sequence of states, and a system is specified by a set of possible behaviors. Thus a behavior is not an event structure. Even though the notion of event appears (an action is a pair consisting of two states in a behavior one following the other), events are not arguments of functions or predicates, and higher-level events do not appear as legitimate objects of the behavior.
2. The Abstract State Machine (ASM) approach introduced by Y. Gurevich (and developed by many researchers) is also based on the notion of Tarskian structure. A state is a structure (an algebra in fact) that contains the truth values as elements of its universe. Yet a "run" is a sequence or a partially ordered set of states—it is not a Tarskian structure (it is not an ASM either). Global states are used in the ASM approach, and this may create difficulties (so it seems to me) when dealing with concurrency issues. For example, several paper investigate a version of the Bakery algorithm and prove its correction under the ASM paradigm [4,5,8]. That Bakery algorithm version uses a single register per process, and it turns out that the algorithm would be incorrect if this register were safe. (I do not know if the ASM approach can deal with safe registers, but I tend to think that it would do so with great difficulties[2]).
3. There are several *event-based* approaches that are similar in spirit to the approach taken here with its emphasize on the events as the prime members of the structure. Pratt [21] uses pomsets (partially ordered multisets) as models, and his modeling approach is quite similar to Lamport's system execution [17] which is the basis for our work here. Another event-based modeling approach which is similar in its spirit to our's is in the work of Constable and Bickford [7], and it seems quite reasonable that the proof in our paper can be formalized in the Nuprl implementation of the theory of event structures referred to in [7] or in one of the HOL proof systems for verification.

2. Tarskian system executions

This section deals with some very fundamental issues: it defines the event structures that are used here to model executions of algorithms. A distributed system consists of n processes $P_1, \ldots, P_n$ communicating with each other with either shared memory or with message-passing channels, and the actions of each P_i are determined by its program (which can also

[2] In an email Y. Gurevich expressed his firm belief that "there is no principled difficulty to handle safe registers by means of global ASMs".

be taken as a finite automaton). In this paper each process is serial, and although the tempo and exact timing of its events are arbitrarily determined, it is active forever.

There are two grand modes of specifying the behavior of such a system: either with a single structure that describes the manifold of all possible executions, or with a collection of structures where each structures describes one possible execution and the collection itself represents the manifold that constitutes the semantics of the system. It is the second mode that is taken here.

There are five parts to this section.

1. We first recall the notions of predicate languages and their interpretations which are called here Tarskian structures.
2. In 2.0.2 we define Tarskian system executions. These are the event structures which represent runs of the system. It is with Tarskian system executions that we shall express the similarities between the two algorithms investigated here. We will need event structures that support high-level events as well as lower-level events, and these Tarskian structures are defined in 2.0.3.
3. The notion of a safe register is very important for us since these registers are employed in the Bakery algorithm. We define safe registers in the context of Tarskian system executions in 2.0.4.

   We describe two different semantics of program executions. The first is based on global states and the second on local states of the participating processes. The first is appropriate for serial registers and for message passing, and the second is appropriate for non-atomic actions such as read and write of safe registers. The following two items 3 and 4 refer to these approaches.
4. In 2.1 we describe the state-history modeling approach in which global states are used. This is a standard approach (see for example [3] or any other distributed computing textbook), and a main reason for bringing this material here is to present the terminology that we shall use. A history (also called run) is a sequence of global state, and as such it is not a Tarskian structure. In 2.1.1 we show how to obtain a Tarskian system execution out of a history of global states. This method is suitable for atomic operations.
5. Since the Bakery algorithm uses non-atomic operations on safe registers, and as global states are less suitable for safe registers, we propose in 2.2 an alternative approach which uses local states to describe each process and its events. This approach is not standard; it is described in details in [2], but the short exposition in 2.2 suffices for our present aim.

We begin with the definition of a multi-sorted predicate language (also called quantification language).

2.0.1. Predicate languages and Tarskian structures

A predicate language is defined by listing its vocabulary, which is technically called its "signature". In order to define a signature for a multi-sorted language one has to make a list of the following items.

1. Names of sorts. The sorts are the types of objects that populate the universe of any interpretation of the signature.
2. Names of variables. The signature can associate variables with specific sorts. For example, if we have a sort named *Event*, one can stipulate that e is an *Event* variable. This allows one to form shorter quantified formulas. For example, $\exists e\varphi(e)$ would mean: there exists an *Event* e such that $\varphi(e)$ holds. It is also possible to use sorts as predicates and to write that statement as $\exists x(Event(x) \wedge \varphi(x))$. This is in fact a very familiar usage in mathematics: when the lecturer writes "for every $\epsilon > 0$ there is K such that for all $k \geq K$ etc." it is immediately understood that ϵ is a real number while K and k are integers.
3. Names of relations (these are the predicates). And for each predicate its arity (number of entries) is given. The signature can also determines the sort of the kth entry of a predicate. For example, predicate $\rightarrow$ has arity 2 (it is a binary predicate) and both of its arguments are of sort *Event*.
4. The signature lists names of functions, and their arities. The signature can also stipulate the sorts of the function entries (parameters) and the sort of values taken by the function. For example, we will have in our system execution signature unary functions called *begin* and *end* which apply to events and take values of sort *Moment*. (The intention is that $begin(e)$ denotes the moment when event e begins and $end(e)$ its ending.)
5. The signature lists constants and their sorts.

Given any signature, there is a standard inductive definition of the resulting *language*, which is the set of all expressions and formulas built with quantifiers and connectives from the elements of the signature.

A Tarskian structure M is an interpretation of a signature; it consists of:

1. A universe, namely a non-empty set A of elements. Usually the universe of M is denoted $|M|$.
2. For each sort S in the signature a set $S^M \subseteq A$ is associated which represents the members of M of sort S.
3. For every relation symbol R of arity k in the signature, a relation $R^M \subseteq A^k$ is associated. In fact, if the signature stipulates that the ith entry of R is of sort S_i then we have $R^M \subseteq S_0^M \times \cdots \times S_{k-1}^M$.
4. For every function symbol F in the signature, of arity k, F^M is a function from A^k to A. Again, it has to respect the stipulations made by the signature about the domain and value sorts of the function.
5. For every constant c of sort S in the signature, $c^M \in S^M$.

Given an interpretation M of a signature and a sentence φ in its language, we say "M satisfies φ" when φ holds true in M (a formal and natural definition can be found in any logic textbook).

Since the names "structure" and "interpretation" have so many different meanings (especially in computer science) we should specifically say "first-order interpretation" or "Tarskian structures" to refer to the structures defined above. We prefer *Tarskian structures* since it was Alfred Tarski who introduced and used them for modeling purposes, and since the term "first-order structures" (which is usually used) is slightly misleading because it is the language rather than the structure that is first-order. Thus first-order structures are suitable for higher-order languages (HOL) as well, and since these languages can be very useful for correctness proofs, we prefer not to use the term first-order structure which indicates a rather limited range of applications.

2.0.2. Tarskian system executions

For the classification of distributed systems we shall use here a certain family of structures which we call *Tarskian system executions with temporal representation.* The term "system executions" is borrowed from Lamport's [17] in order to acknowledge that this notion is originally due to that work and to emphasize that our main contribution here is to bring it within the larger framework of Tarski's model theory. One of the referees suggested the term Tarskian event structures, and this also makes sense, because it connects this notion to works which view events rather than states as the prime objects. Anyhow, the signature of these structures is defined first.

Definition 2.1. A system execution signature is a signature that contains the following items (and possibly more).

1. There are three sorts: *Event*, *Atemporal*, and *Moment* (there may be additional sorts, but these three are required). *Event* and *Atemporal* are disjoint, *Moment* is a subsort of *Atemporal*.
2. There is a binary relation $<$ on *Moment*. Intuitively, $m_1 < m_2$ means that moment m_1 precedes (is earlier than) moment m_2. We write $a \leq b$ for $a < b$ *or* $a = b$.
3. There is a binary relation symbol $\rightarrow$ defined on *Event*. It is called the temporal precedence relation. Intuitively $e_1 \rightarrow e_2$ means that event e_1 ends before e_2 begins. (In some fields of study the term event refers to a type of event occurrences, but for us an event *is* an event occurrence and it has a definite temporal extension.)
4. There are two function symbols *begin* and *end* denoting functions from *Event* into *Moment*. We think of event e as extended in time and represented by the closed interval of moments $[begin(e), end(e)]$.
5. *terminating* is a unary predicate on *Event*. Intuitively, *terminating*(e) says that event e has a bounded duration.
6. It is convenient to have a constant ∞ in sort *Moment* which serves as the right-end point of non-terminating intervals.
7. There are possibly other predicates, functions, and constants in the signature.

Having defined their signatures, we next define Tarskian system executions.

Definition 2.2. A Tarskian system execution (with temporal representation) is an interpretation M of a system execution signature so that:

1. $Moment^M$ is linearly ordered by $<^M$. ∞^M is the last member of $Moment^M$. (In this paper the order-type of $Moment^M$ is that of the natural numbers with an additional point, the infinity, at the end.)
2. The following hold in M:
   (a) For every event e, $begin(e) \leq end(e)$. The set $[begin(e), end(e)] = \{x \in Moment^M \mid begin(e) \leq^M x \leq^M end(e)\}$ is called the "temporal interval of e".
   (b) For every events e_1 and e_2, $e_1 \rightarrow e_2$ iff $end(e_1) < begin(e_2)$. ($\rightarrow$ turns out to be transitive and irreflexive.)
   (c) For every event e, *terminating*(e) iff $end(e) < \infty$.
3. For every terminating event e there exists a finite set X of events such that if a is any event not in X then $e \rightarrow a$. If e is a non-terminating event then a weaker property holds, namely that the set of events $\{x \mid x \rightarrow e\}$ is finite.

Since $<^M$ is a linear ordering of $Moment^M$, $\rightarrow$ is a partial ordering of $Event^M$. Usually, it is not a linear ordering. We say that events e_1 and e_2 are *concurrent* if $e_1 \not\rightarrow e_2$ and $e_2 \not\rightarrow e_1$.

The third property above is the finiteness axiom of Lamport (introduced in [17]). An equivalent formulation is that for every event e there are only finitely many events x such that $x \rightarrow e$, and if e is a terminating event then only finitely many events are concurrent with e.

Properly speaking, we defined here system executions "with temporal representation" which means that a sort of moments is part of the structure. System executions without an explicit representation of the time axis are also useful, but will not appear in this paper.

2.0.3. System executions with high-level events

As Lamport does in [17] we reason in this paper about high and low-level events and their temporal relationships. A low-level event is for example a write or a read of a register, or a send/receive event. A high-level event is a (non-empty)

set of lower-level events which constitute an operation execution. To formally deal with lower and higher-level events we introduce a membership predicate, $\in$, to the system execution signature, and then relation $a \in E$ for events a and E means that a is a lower-level event that belongs to E which is a high-level event. In system executions with low and high-level events which interpret this richer signature we usually have the following additional requirement.

If E is a high-level event then $begin(E) = \min\{begin(x) \mid x \in E\}$
and in case E is terminating $end(E) = \max\{end(x) \mid x \in E\}$.
If E is a non-terminating high-level event then $end(E) = \infty$.

We can extend the defining property of $\rightarrow$ (2(b) of Definition 2.2) on high and low-level events: $X \rightarrow Y$ if $end(X) < begin(Y)$.

Thus we have the following.

1. For high-level events E_1 and E_2 we have $E_1 \rightarrow E_2$ iff for all $a \in E_1$ and $b \in E_2$ $a \rightarrow b$.
2. Similarly, if a is a low-level event and E a higher-level event, then $a \rightarrow E$ iff for all $e \in E$ we have $a \rightarrow e$. (And the symmetric form holds for $E \rightarrow a$.)

Thus usually a high-level event E is terminating iff it contains a finite number of lower-level terminating events. But if a high-level event E is (for example) an execution of an instruction "wait until α" where α is some condition that never materializes, then in fact E is non-terminating and we define $end(E) = \infty$ even though E may contain a finite number of lower-level.

2.0.4. Specifying safe registers

A register is a shared memory location that supports read and write operations. Lamport [17] defined the notion of a safe register by means of system executions which describe the behavior of read and write events on the register. We redo this definition here but using Tarskian system executions and thus we give a simple example of the usage of system executions. We deal here with "single-writer multi-reader" registers, which means that a specific serial process (the "owner" of the register) can execute write operations on the register, and several processes can access the register for read operations. Thus, whereas the write events are serially ordered in time (the owner of the register being a serial process) the read events are not, and two read operations by different processes may be concurrent.

Intuitively speaking, a register is safe when the write events are serially ordered, and every read returns the value of the rightmost write preceding it, provided that the read is not concurrent with any write. So a read that is concurrent with a write returns an arbitrary value from the range of values associated with the register.

First a signature L_{safe} for the register language is defined.

Definition 2.3. Let R be a name (intended to be a register name). A "register signature for R" is a system execution signature (as in Definition 2.1) that contains the following features. [In square brackets we write some comments and intuitive explanations.]

1. Sorts are *Event* and *Moment* which are in any system execution signature, and the *Data* sort [used to represent the set of register values].
2. A partial function *Value* is defined on *Event* and takes its values in *Data*. The functions *begin*, *end* and the predicate *terminating* are also present as in any other system execution signature. [The function *Value* is partial since it is defined only on read/write events and not on other events which may possibly be in the structure.]
3. Two unary predicates are defined on sort *Event*: $Read_R$ and $Write_R$. [The intention is to use $Write_R(e)$ to say that event e is a write event on R, and $Read_R(e)$ to say that e is a read of that register. When the identity of R is obvious we may write *Write* and *Read* without the specific reference to R.]
4. As we said, we define here a system execution signature and hence all features of such signatures appear here as well, and in particular a binary predicate $\rightarrow$ is defined over the *Event* sort.

Definition 2.4. Assume that S is a Tarskian system execution that interprets a register signature for R (the name of a register). We say that "R is safe in S" if the following hold in S.

1. Every read/write event is terminating. The set of write events on R in S (namely the extension of the predicate $Write_R^S$) is linearly ordered by $\rightarrow^S$.
2. There exists a write onto R that precedes all reads in the temporal precedence relation $\rightarrow^S$. This first write is said to be the initial write.
3. For every read event r either r is concurrent with some write w (i.e., $\neg((r \rightarrow w) \vee (w \rightarrow r))$), or else $Value(r) = Value(w)$, where w is the $\rightarrow$-rightmost write on R such that $w \rightarrow r$. That is, the value returned by r equals the value written by the latest write preceding it.

Observe that the existence of the rightmost write on R that precedes r follows from Lamport finiteness property and the assumptions that an initial write exists and that the write events are linearly ordered.

We shall denote with $\mathcal{S}^R_{\text{safe}}$ the class of all system executions S in which R is safe (as defined above).

For correctness proofs of algorithms that use safe registers we find it useful to have a "reading from" function ρ defined on any read event r and returning the corresponding write event whose value r returns. The following lemma whose proof is quite simple gives the details of the definition of ρ.

Lemma 2.5. *Suppose that S is a system execution that interprets a register signature for R, so that the read/write events on R are terminating, the write events on R are linearly ordered (in the temporal precedence relation $\rightarrow$), and the first write precedes all reads. Then R is a safe register in S (Definition 2.4) if and only if S can be expanded by defining a function ρ over the read of R events and returning write on R events so that the following hold for every read event r:*

1. $r \not\rightarrow \rho(r)$.
2. *There is no write w onto R such that $\rho(r) \rightarrow w \rightarrow r$.*
3. *If $\rho(r) \rightarrow r$, then $Value(r) = Value(\rho(r))$.*

Proof. We shall outline the proof that being safe as in Definition 2.4 entails the three properties of the lemma. Given a system execution S in which R is a safe register, we define the function ρ over the read of R events as follows. If read r is concurrent with some write then we let w be any such write and define $w = \rho(r)$. (Concurrency of two events is defined immediately after Definition 2.2.) Otherwise, for every write on R, w, we have $w \rightarrow r$ or $r \rightarrow w$. By the finiteness axiom, the set of write events w such that $w \rightarrow r$ is finite and non-empty (by the existence of an initial write). Since this set is linearly ordered under $\rightarrow$, we can define $\rho(r)$ as its last event. The three properties follow directly from the definition of ρ and the requirements of Definition 2.4. □

2.1. States and histories

Not only it is often difficult to prove the correctness of an algorithm, but it is not always clear what it means that the algorithm is correct. Given an algorithm A and a correctness formula φ in a language L, we must answer the following two questions.

1. How can an execution of A be represented as a structure for the language L?
2. Is it indeed the case that M satisfies φ whenever M is a structure that represents an execution of A?

There is a standard answer to these questions which is adopted by the distributed computing community and which has proven its value both in industry and theory: that is the state and history approach. We outline this approach in the following paragraphs in order to explain why it does not fit our needs, and in Sections 2.2 and 2.1.1 we describe two different answers which will be used here—one for atomic operations and the other for safe register communications.

Given a distributed system, a "global state" is a description of an instant in an execution. In many cases, this description is given by a function which assigns values to the system variables (or state variables as we shall call them), to each variable a value in its type.

Then we have the notion of a "step" which is a pair of states $s = \langle \sigma_1, \sigma_2 \rangle$ that represents an atomic event obtained when one of the processes executes an atomic instruction in its algorithm. For example, if step $\langle \sigma_1, \sigma_2 \rangle$ represents an execution of an instruction WRITE$(R := v)$ by process P_i (an atomic write on register R of the value of v) then the program counter of P_i points at state σ_1 to that instruction, and points to the next instruction in the code in state σ_2. The value of R at state σ_2 is the value of v at σ_1.

A "history" (also called a run) can be defined as a sequence of states (finite or infinite) $\sigma_0, \sigma_1, \ldots$ so that σ_0 is an initial state and each pair $\langle \sigma_i, \sigma_{i+1} \rangle$ is a step by one of the processes. Concurrency is obtained by interleaving the steps (atomic actions) by the different processes. Of course, there is much more to say about states, steps and histories (for example about fairness), but our aim in this short description is to establish a common terminology to serve us later on.

A history, defined as a sequence of states, is not a Tarskian structure, namely there is no logical quantification language that it interprets, and so it is not suitable for us. It is however a temporal logic structure and it gives truth values to temporal logic formulas (see for example [20,18]). For the purpose of classification, Tarskian structures have an advantage, and a main aim of this article is to explain this advantage. So in the following subsection we show how a history of global states can be transformed into a Tarskian system execution.

2.1.1. From global state histories to Tarskian system executions

Given a history $r = (\sigma_i \mid i \in \omega)$, we outline here the process of making a Tarskian system execution denoted $H(r)$ out of r. (For uniformity and simplicity, we deal with infinite histories, and ω is the set of natural numbers.) Recall Definition 2.2: we have to define the universe of $M = H(r)$ and the different interpretations of relations and functions.

1. The main elements of the universe of M are its low-level events formed from the steps of r; each step is a low-level event. Then the finite sets of events that represent operation executions in r are the high-level events of M. Together they form the $Event^M$ sort.
2. The set of natural numbers together with an infinity point form the $Moment^M$ sort. The ordering $<^M$ of this sort is the ordering of the natural numbers together with the relations $n <^M \infty$ for every natural number n.
3. The different types that the algorithm requires are sorts of M (part of *Atemporal*). For example, we have a sort of values that the different registers can have.

4. Functions and predicates in M are needed to characterize the steps. For example, suppose that step $e = \langle \sigma_i, \sigma_{i+1} \rangle$ is a write event of value v done by process P_i on its register R_i. Then we will have $Write^M(e)$ (where *Write* is a unary predicate), $P_i^M(e)$ (where P_i is a predicate), and $address^M(e) =$ "R_i" is the location (register name) on which the write event occurs.
5. The functions *begin* and *end* are defined on the low-level events first. There is an infinite number of possibilities to define these functions. The simplest is the following. If $e = \langle \sigma_i, \sigma_{i+1} \rangle$ is any event in M, then we can define $begin(e) = end(e) = i$. But any other definition is possible if it satisfies the following: for every $e = \langle \sigma_i, \sigma_{i+1} \rangle$ and $e' = \langle \sigma_{i+1}, \sigma_{i+2} \rangle$: $begin(e) \leq end(e) < begin(e')$.

Now higher-level events can be introduced to M as in 2.0.3.

This ends the definition of the event structure $H(r)$ that corresponds to a history r. Given any correctness statement φ we can ask whether $H(r)$ satisfies φ or not—this is now a well-defined relation, and to say that an algorithm is correct is to say that $H(r)$ satisfies φ whenever history r is an execution of the algorithm. This type of modeling with histories that are based on global states works well for serial registers and for message-passing communication. In general, whenever the basic communication events can be thought of as occurring in an instant of time (as opposed to an interval), global states, histories of executions, and the resulting Tarskian event structures are suitable. But when the read and write events (or any other communication events) are extended in time, then a single step cannot be a substitute for the interval in which the event occurs. It is possible of course that a step refers to the beginning of the event and a later step to its end (this is the approach taken for example by [10]), yet it would be quite cumbersome to handle safe registers this way, and in the following section we outline an alternative approach that does not rely on global states.

2.2. Tarskian structures with local states

In this section we define the semantics of programs that use communication means such as safe registers for which global states and atomic steps are inadequate. We prefer to use local states to describe the operations of each process and to combine these descriptions with the specification of safe registers (using Tarskian system executions as in Section 2.0.4).

Recall how safe registers were defined in that section. We defined a certain register signature L_{safe} (Definition 2.3) and a certain class $\mathcal{S}^R_{\text{safe}}$ of safe register structures which represent correct behavior of the read/write events (Definition 2.4). That definition uses no states at all, and relies on the partial ordering of the read/write events and their values.

Now, the definition of the set of all executions of a system consisting of serial processes that communicate with safe registers is done in two stages. In the first stage we define those Tarskian system executions that describe executions that have no restriction whatsoever on the values that the read actions return. Then in the second stage we impose on these non-restricted system executions the requirements of safe registers as defined in Section 2.0.4, and obtain in this fashion the required set of restricted executions. For the first stage only local states of the processes are required, but no states are used in the second stage. An outline of the first stage is as follows.

Let M be a Tarskian system execution with N serial processes. Some events of process P are read events and some are write events, and every event has a value. How can we determine that these events in M follow the program that P is executing? A simple but important observation is that the answer to this question depends only on the local states of P and not on the other processes or on the semantics of the communication media employed by P. To describe the answer in some details we first define local histories of process P by means of its local states.

A local state of process P gives information on P's local variables, but nothing on the shared registers and their values. The program of P can be described as a deterministic automaton whose nodes are the local states of P. We assume a set of *actions*; each action a has an attribute (e.g. $read(a)$, $write(a)$), a value $Val(a)$, and an address $node(a)$ which is a local state of P on which this action operates. An action a is enabled at $node(a)$ and (for simplicity) it is not enabled at other states. If a is enabled at local state σ then $a(\sigma)$ is the local state that results when action a is taken. Thus $\langle \sigma, a(\sigma) \rangle$ is the step that represents an execution of this action by P.

Consider for example read/write actions. Suppose that a read action of register R is enabled at state σ. That is, the process is required to execute instruction READ($x := R$). Then for every value v in the range of R a read action a with value v is enabled at σ, and the value of variable x at the resulting state $a(\sigma)$ is v. Note that there is no restriction on the value of the read action and any value v is a possible outcome because the local state σ does not record the value of the global register R. If at state σ our process P is required to execute a write instruction WRITE($R' := t$) on register R', then a write action a is enabled and its value is the value of t at σ. So a write action does not change the value of a register (since the register is not part of the local state), and the value of a read action does not depend on the state (since the value of the register read is not part of the state).

A *local run* of process P is defined to be a sequence $r = (\sigma_0, a_0, \sigma_1, a_1, \ldots)$ of local states σ_i alternating with local actions a_i of P so that $a_i(\sigma_i) = \sigma_{i+1}$. A local run is thus a history of a single, isolated process in the system. Again: knowing the program of P we can define its local runs in isolation from any runs of the other processes and their shared communication devices. From the point of view of the executing process P, only the values returned by calls to these devices count—not the reasons for their values. Most of these local runs will be uninteresting in the sense that it would be impossible to see them as part of a system execution which involves all processes, but nevertheless this simple notion of local runs is an important building block in our local states based semantics.

l_1 WRITE($G_i := 1$)
l_2 $n :=$ *take-a-number*
l_3 WRITE($R_i := n$)
l_4 WRITE($G_i := 0$)
l_5 **forall** $j \neq i$
  1. **repeat** READ($g := G_j$) **until** $g = 0$
  2. **repeat** READ($k := R_j$) **until** $(k, j) > (n, i)$
l_6 Critical Section
l_7 WRITE($R_i := \infty$).

Fig. 1. Take-A-Number variant of the Bakery protocol for process P_i which alternates protocol executions with its Remainder sections. Registers G_i and R_i are safe. Here $(k, j) > (n, i)$ is a shorthand for $k > n$ or else $k = n$ and $j > i$ (lexicographical ordering). The value of k can also be ∞, in which case $k > n$ for any natural number n. The initial value of G_i is 0, and that of R_i is ∞.

Returning to the question posed above (how to determine that the events in M attributed to P follow its program) we can say now that they do if there exist a local run r of process P, $r = (\sigma_0, a_0, \sigma_1, a_1, \ldots)$, and an order preserving injection f of the set of actions $\{a_i \mid i \in \omega\}$ into the events of P in M so that the attributes of every action a_i are retained by the predicates and functions of M as applied to the event $f(a_i)$. For example, if a_i is a reading step of value $v = Val(a_i)$ and $e = f(a_i)$ is the corresponding event in M, then $Val^M(e) = v$ and $Read^M(e)$. If every process in M follows its program, then we say that M is a non-restricted system execution. It is non-restricted in the sense that the values returned by its reading events are completely arbitrary.

Note that whereas step (σ_i, σ_{i+1}) is, from the point of view of the executing process, an atomic step, the corresponding event $e = f(a_i)$ of P in M has a temporal extension $(begin(e), end(e))$ whose length is not determined by P.

Another important point is that global states were not used to define non-restricted Tarskian system executions. Since register R is not a local variable of any process, no local state refers to R and its values. In a non-restricted system execution we have read/write events, their values and their temporal relations, but the register itself is not explicitly represented in the structure (unlike the global state based history in which R is a variable that has a value in each state and state).

Now if M is a non-restricted system execution as defined above, and it happens that the read/write events by the different processes and their values satisfy the specification of safe registers as defined in 2.0.4, then M is defined to be a *restricted execution*. In a restricted system execution not only each process works in accordance with its program, but the register operations respect the safe register specifications as well. In other words, a restricted system execution has to satisfy two types of requirements: that each process behaves as its program dictates (and for this only local states of the program are needed) and that the communication devices behave correctly (and this is expressed in the language of the system execution and not by means of any global states).

We are ready now for the Take-A-Number safe register algorithm.

3. The Take-A-Number algorithm

The Take-A-Number algorithm (Fig. 1) is a critical section algorithm that is a variant of Lamport's Bakery mutual-exclusion algorithm [12]. This variant (first presented in [1]) simplifies the procedure by which a process chooses a timestamp, but we have kept the most intriguing feature of Lamport's original Bakery algorithm—the usage of safe registers. Assuming that the registers are serial would over simplify the algorithm to the point of making it much less interesting for us.

In the Take-A-Number algorithm each process calls a function called *take-a-number*, which returns a natural number, and line l_2

$n :=$ *take-a-number*

assigns this number to (local) variable n. To ensure the correct functioning of the protocol we assume that:

> if r_1 and r_2 are invocations of the *take-a-number* function (possibly by different processes) and if r_1 precedes r_2, then the value returned by r_1 is strictly smaller than that of r_2. (If r_1 and r_2 are concurrent, then no assumption is made on the relation between the values returned.) (1)

The algorithm is designed for N serial processes $P_1, \ldots, P_N$. Each process P_i uses safe 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, that is its domain of values is $\{0, 1\}$, and it is initially set to 0. The domain of register 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 initial event that writes the initial value).

In the algorithm, WRITE($G_i := v$) is the instruction that writes v on register G_i. We prefer this format over the simpler $G_i := v$ in order to emphasize that this is not supposed to be an atomic event. Similarly, READ($g := G_j$) is the instruction to read register G_j and to assign the value returned to local variable g.

For the reader who is less familiar with the Bakery algorithm we describe an example that illustrate the need for the G_i registers in the Take-A-Number (and the Bakery) algorithm. Suppose that the writes and reads related to registers G_i are removed from the algorithm. Then the mutual-exclusion property does not hold even when the registers are serial. Indeed consider the case of just twp processes P_0 and P_1. The initial values of their registers (R_0 and R_1) is ∞. Say P_0 begins and executes the *take-a-number* instruction obtaining, say, the value 0. It intends to write that value on its R_0 register, but is delayed for a moment during which P_1 obtains 1 in executing its *take-a-number* instruction, writes this number on register R_1, reads ∞ in register R_0 and enter its critical section as its protocol says. Only then P_0 returns to work, writes the value 0 on R_0 and reads 1 from register R_1 of P_1. This value allows P_0 to enter its critical section, thereby violating the mutual-exclusion property.

With the G_0 register and the writes at lines l_1 and l_4 of the values 1 and 0 this particular scenario is impossible. For P_1 would have to read the value 0 in register G_0 before being allowed to access register R_0. Of course, one needs a proof not an elimination of a particular counterexample, and it is in Lemma 3.2 that the value of registers G_i is demonstrated.

In order to be able to formulate our main lemmas (3.2 and 3.3), we need some definitions.

Definition 3.1. Consider an execution of the Take-A-Number algorithm.

1. The events are categorized as low-level and high-level [17]. Low-level events are the read/write events, executions of take-a-number, and critical section events. High-level events are executions of the protocol. A "protocol execution" is either an execution of lines l_1 through l_7 of the protocol of Fig. 1 (this is a "terminating" execution) or an execution of lines l_1 through l_5 in which a **repeat** loop is non-terminating (this is a "non-terminating" execution). A protocol execution is a high-level event represented as a set of lower-level events.

   A terminating execution is 'successful' as it succeeds to enter its critical section.
2. If e is any event, high or low level, then there is a process P_i to which e "belongs". We view P_i both as a set—the set of all events that belong to P_i, and as a predicate that holds for those events that belong to P_i. We assume that each P_i is a serial process.
3. The temporal relation $a \to b$ (meaning a ends before b begins) is a partial ordering of the events.
4. An execution of line l_2 by one of the processes is called a "take-a-number" event. These events are extended in time (and like read and write events they always terminate).

   With each take-a-number event e we associate the pair $(n, j) = Val(e)$ where n is the value (a natural number) returned by this action and j is the index of the executing process P_j. If take-a-number events e_1 and e_2 are associated with (n_1, i_1) and (n_2, i_2) respectively, then we say that e_2 "dominates" e_1 if either $n_2 > n_1$ or else $n_2 = n_1$ and $i_2 > i_1$. That is, e_2 dominates e_1 if $Val(e_1) < Val(e_2)$ lexicographically.

   Let x and y be two take-a-number events such that $y \to x$. By assumption (1), the value returned by y is strictly smaller than the value returned by x, and hence x dominates y in this case.

   It is obvious that if $e_1 \neq e_2$ are take-a-number events then the pairs $(n_1, i_1) = Val(e_1)$ and $(n_2, i_2) = Val(e_2)$ are different, and hence either e_1 dominates e_2 or else e_2 dominates e_1. (If $i_1 \neq i_2$ then clearly it is the case that $Val(e_1) \neq Val(e_2)$, but otherwise we rely on the assumption that P_i is a serial process and hence $i_1 = i_2$ implies that either $e_1 \to e_2$ or $e_2 \to e_1$, so that $n_1 \neq n_2$ follows from our assumption displayed in Eq. (1)).
5. Suppose that X is a protocol execution (terminating or not). Then $take(X)$ denotes the take-a-number event of X. Namely the lower-level event in X that is the execution of line l_2.
6. If protocol execution X is terminating, then it contains a critical section event denoted $cs(X)$. Any critical section event is of the form $cs(X)$ for some terminating protocol execution X (no process executes a critical section unless it successfully executes the Take-A-Number protocol).
7. Suppose that X is a terminating protocol execution by process P_i. Then, for every index $j \neq i$, X contains a successful execution of instruction $l_5(2)$ for index j. That is, an execution of $\text{READ}(k := R_j)$ that obtains a value k such that condition $(k, j) > (n, i)$ holds. (k is either a natural number or ∞, and n is always a natural number being the value returned by the *take-a-number* procedure.) Let r_j denote this successful read of register R_j. (X also contains a successful read of register G_j, but the read of R_j somehow plays a more prominent role in the correctness proof.) Using, the semantics of safe registers (Lemma 2.5), we let $\rho(r_j)$ be the corresponding write onto register R_j. So $\rho(r_j)$ is either concurrent with r_j, or else $\rho(r_j) \to r_j$ and the value of r_j equals that of $\rho(r_j)$. In any case there is no write w_1 on R_j such that $\rho(r_j) \to w_1 \to r_j$. Now, like any other write onto register R_j, $\rho(r_j)$ is either in some unique protocol execution or else it is the initial write onto register R_j (a write of ∞). In the first case we define $\Omega(X, j)$ to be that protocol execution (a high-level event) that contains $\rho(r)$, and in the second case $\Omega(X, j)$ is the initial write on R_j.

   In case $\Omega(X, j) = E$ is a protocol execution, then $\rho(r) \in E$ is either the first write in E on register R_j, or else the second write (of value ∞).

   Intuitively speaking (and forgetting for a moment the possibility that when r and $\rho(r)$ are concurrent their values need not be the same) $\Omega(X, j)$ is that high-level event of index j that gave X the permission to continue, either because $\rho(r)$ is a write of value ∞ or else because $\langle Val(\rho(r)), j\rangle$ lexicographically dominates the pair $\langle n, i\rangle$ where n is the number taken by X in executing line l_2.

We formulate next two lemmas and prove that they hold in any system execution of the Take-A-Number protocol. Then we prove (Theorem 3.4) that they entail the mutual-exclusion property. These lemmas have a prominent place in the abstract high-level proof given in the following section.

It turns out that these lemmas require only safe registers, and this is rather surprising: since a read concurrent with a write can return an arbitrary value, the reader, so it seems at first, cannot get any information from its read of a safe register. Indeed, since that read may be concurrent with some write and the reader has no way of knowing this fact, how can its value be informative? Still, the proofs show how a clever code can use two safe registers to overcome their limitations.

Lemma 3.2. *Suppose that X and Y are protocol executions, by processes P_j and P_i respectively where $i \neq j$. Suppose that Y is terminating and $\Omega(Y, j) \to X$. Let $y = take(Y)$ and $x = take(X)$ be their respective take-a-number events.* **Then** $y \to x$.

Proof. Assume on the contrary that this is not the case and $y \to x$ does not hold. If the *take-a-number* events were serially ordered, we would be able to deduce that $x \to y$ is the case (as $x \neq y$), but since we have no such assumption we can only get that

$$begin(x) \leq end(y). \tag{2}$$

We will use the following events that must exist in Y (being a terminating protocol execution). In addition to $y = take(Y)$, we let s and r be the successful reads in Y of registers G_j and R_j. That is, s is that execution in Y of the read in $l_5 1$ of register G_j that obtained the value 0, and r is the successful read of R_j in Y that corresponds to $l_5 2$. Clearly $y \to s \to r$. By the definition of Ω, $\Omega(Y, j)$ is either a protocol execution by P_j so that $\rho(r) \in \Omega(Y, j)$ or else $\rho(r) = \Omega(Y, j)$ is the initial write on register R_j. We shall prove that the write w in X that executes line l_3, namely the first write on R_j in X, is such that

$$w \to r.$$

But then the semantics of safe registers (namely item 2 of Lemma 2.5) imply that $w = \rho(r)$ or $w \to \rho(r)$. But this implies (as $w \in X$) that $X = \Omega(Y, j)$ or $X \to \Omega(Y, j)$, and both possibilities contradict $\Omega(Y, j) \to X$.

Now, to prove that the write w in X corresponding to l_3 satisfies relation $w \to r$, we observe that the write event v in X of value 1 on register G_j precedes x (which is its take-a-number event), and so we have $v \to x$, $begin(x) \leq end(y)$, and $y \to s$, which imply that $v \to s$. But v is a write of value 1 and s a read of 0. Let v' denote the write of 0 in X on G_j following v (an execution of ℓ_4). So $v \to w \to v'$. Surely, $s \to v'$ is impossible (as in this case $v \to s \to v'$ and s would return 1), and hence $begin(v') \leq end(s)$. But $w \to v'$, and as $s \to r$ we get $w \to r$ as required. □

Lemma 3.3. *Assume that Y is a terminating protocol execution by P_i, and $\Omega(Y, j) = X$ is a protocol execution by P_j. Let $x = take(X)$ and $y = take(Y)$. Then*

1. *$x \to cs(Y)$, and*
2. *x dominates y, or X is terminating and $cs(X) \to cs(Y)$.*

Proof. Let r be the successful read in Y of register R_j (obtaining a value k so that requirement $(k, j) > (n, i)$ holds). Say $w = \rho(r)$ is the corresponding write on that register. By definition of Ω and the assumption that $\Omega(Y, j) = X$ is a protocol execution (rather than the initial write), we get that $w \in X$. So w is either the write w_0 on R_j that corresponds to line l_3 or the write w_1 that corresponds to l_7. In any case $x \to w$, and as $r \not\to \rho(r)$, and $r \to cs(Y)$, a short argument yields $x \to cs(Y)$. This proves 1.

To prove item 2, consider the two possibilities for w in X.

Case A: $w = w_1$ is the execution of l_7, namely the write of ∞ on R_j. In this case, X is terminating. We have $cs(X) \to w_1$, $r \not\to \rho(r) = w_1$, and $r \to cs(Y)$. Hence $cs(X) \to cs(Y)$ can easily be concluded.

Case B: $w = w_0$ is the execution of l_3 in X. We will prove that in this case x dominates y. If $y \to x$, then x dominates y by assumption (1) on take-a-number executions, and hence item (2) of the lemma holds in this case. So we assume that $y \not\to x$, namely that

$$begin(x) \leq end(y). \tag{3}$$

Let $v_1 \to v_0$ be the writes on G_j in X of values 1 and 0 respectively corresponding to lines l_1 and l_4. We have

$$v_1 \to x \to w_0 \to v_0$$

by the protocol instruction ordering. Let $s \in Y$ be that read of register G_j that obtained the value 0 in executing line $l_5(1)$. So

$$s \to r.$$

Since $y \to s$ and $v_1 \to x$, it follows from (3) that $v_1 \to s$. If $s \to v_0$, then $v_1 \to s \to v_0$, namely the read s is strictly between the write v_1 and its successor write v_0 and hence the value of s is 1 (the value written by v_1) which is not the case. Hence $s \not\to v_0$, namely

$$begin(v_0) \leq end(s).$$

As $w_0 \to v_0$, and $s \to r$, $w_0 \to r$ follows. Yet $w_0 = \rho(r)$. So $Val(r) = Val(w_0)$ (by the specification of safe register, see item 3 of Lemma 2.5). This immediately implies that x dominates y, since condition $(k, j) > (n, i)$ on line $l_5(2)$ holds. □

We are now ready to prove the mutual-exclusion property of our algorithm. As the reader will notice, the proof uses mainly the last two Lemmas 3.2 and 3.3.

Theorem 3.4. *Suppose that X and Y are two terminating protocol executions by P_j and P_i respectively where $i \neq j$. Then $cs(X)$ and $cs(Y)$ are not concurrent.*

Proof. Say $x = take(X)$ and $y = take(Y)$. Then either x dominates y or else y dominates x (item 4 of Definition 3.1). Suppose without loss of generality that y dominates x. This implies that

$$\neg(y \to x) \tag{4}$$

by assumption (1). Now consider $\Omega(Y, j)$. Since X is a protocol execution in P_j and $\Omega(Y, j)$ is either a protocol execution in P_j or the initial event there, we have one of the following three possibilities.

1. $\Omega(Y, j) \to X$. Then Lemma 3.2 applies and yields that $y \to x$, in contradiction to (4).
2. $\Omega(Y, j) = X$. We apply now Lemma 3.3. Since y dominates x it is not the case that x dominates y, and item 2 of that lemma implies that $cs(X) \to cs(Y)$. So these two critical section events are not concurrent.
3. $X \to \Omega(Y, j)$. Say $\Omega(Y, j) = Z$. Then by Lemma 3.3 item 1, $take(Z) \to cs(Y)$. But $X \to Z$ clearly implies that $cs(X) \to take(Z)$, and hence $cs(X) \to cs(Y)$ and these critical section events are again not concurrent.

3.1. Discussion

Lemmas 3.2 and 3.3 have a prominent role in this paper. We have seen that they serve in Theorem 3.4 to conclude the mutual-exclusion property of the Take-A-Number algorithm, and we will see in the last part of our article that any execution of the Ricart and Agrawala algorithm also satisfies the properties that these lemmas establish. Hence it makes sense (and this is a very common practice) to formulate these properties abstractly, and to prove that they imply the mutual-exclusion property directly and without any reference to a specific algorithm. This is done in Section 4 and the list of properties in Fig. 3 encapsulates our two lemmas in the general setting of system executions. The abstract mutual-exclusion proof and the demonstration that the correctness of the two algorithms can be obtained by means of this abstract proof certainly highlight the importance of these lemmas. But we make a stronger claim about their role: namely that they answer our quest of formalizing the intuitive reasons for the similarities between the two algorithms. Just exposing a common part of the correctness proofs and distilling that part in an abstract setting may be seen as a contrived demonstration of mathematical bravura rather than a deeper analysis of their similarities.

We therefore need an informal and intuitive discussion that argue how Lemmas 3.2 and 3.3 (or rather their subsequent abstract formulation) reflect an intuitive understanding of a subfamily of the Bakery like algorithms to which the Take-A-Number and the Ricart and Agrawala algorithms belong. The idea in all Bakery like algorithms is that in order to enter its critical section, process P_i has to get "permissions" from all other processes, and that these permissions are based on some evidence of priority that a certain class of "timestamps" provide. This priority relation is formalized in our lemmas and abstract properties by the "dominance" relation. Given protocol executions X and Y, X dominates Y means intuitively that X came later than Y and thus Y is entitled to enter its critical section before X. In every protocol execution X we identify an early part $take(X)$ in which X obtains its timestamp. Surely $take(X)$ must be located at the beginning of X since that timestamp is used later on to give and take entry permissions. The functional relation $X = \Omega(Y, j)$ is of prime importance. Whenever Y is a terminating protocol execution by P_i (namely an execution that contains a successful entry to its critical section) relation $X = \Omega(Y, j)$ expresses the fact that X is the protocol execution (or initial event) by P_j that gave a permission to Y to enter its critical section. Whenever Y has obtained permissions from all other processes it may enter its critical section, There are two types of entry permissions that a process can grant:

1. When a protocol execution X has terminated and is in its subsequent remainder (external) phase, it gives an open permission to all other processes to enter their critical sections. In the Take-A-Number algorithm this open permission is given by setting register R_j to ∞, but in the Ricart and Agrawala algorithm process P_j immediately sends "OK" messages as a response to any request it obtains while at the "Remainder" phase. In case Y (an execution by P_i) obtained permission from P_j while P_j is at its remainder phase following an execution X that had terminated, we have $X = \Omega(Y, j)$, and item 2 of Lemma 3.3 expresses this possibility with the clause "X is terminating and $cs(X) \to cs(Y)$. This is intuitively obvious: X has terminated since P_j is at its remainder phase, and as Y which is still collecting permissions is not yet at its critical section, $cs(X) \to cs(Y)$ follows.
2. It is also possible that P_j gives its permission to P_i while P_j is still executing its protocol (before it got into its critical section) since it realizes that P_j dominates P_i. This corresponds to clause "x dominates y" in item 2 of Lemma 3.3.

This shows that item 2 of Lemma 3.3 is a natural expression of these two types of permissions that Y can obtain. But item 1 is also very natural: If $X = \Omega(Y, j)$, then $x \to cs(Y)$ where $x = take(X)$. Indeed X gives permissions only after its take-a-number event since these permissions are based on the value of that event, and Y enters its critical section only after getting the permission, so that $x \to cs(Y)$ follows.

Lemma 3.2 may appear less natural and we shall justify it by explaining why its negation

$$Y \text{ is terminating, } (\Omega(Y, j) \to X), \text{ and } \neg(y \to x)$$

A: For every event e there is one and only one j ($1 \leq j \leq N$) such that $P_j(e)$. j is called the "process index" of e (and we say that e "is in P_j"). If $PE(X)$ then the process index of X is the same as the process index of $take(X)$ and of $cs(X)$ (when $cs(X)$ is defined).

B: Relation $\prec$ is irreflexive and transitive over the events. The seriality property is that for every process index j the set of protocol execution events in P_j are linearly ordered under $\prec$. That is, if $P_j(X)$ and $P_j(Y)$ and $PE(X)$ and $PE(Y)$, then $X \prec Y$ or $Y \prec X$ or $X = Y$. For events E_1 and E_1 that are both in P_j $E_1 \prec E_2$ iff $end(E_1) < begin(E_2)$ (iff $E_1 \to E_2$).

We assume an "initializing event" in P_j (it is not a PE event) which $\prec$-precedes all PE events by P_j.

C: The following functions are partial and the following properties define their domains.

1. The domain of $take$ is the set of all protocol execution events. If $x = take(X)$ we say that x is a take-a-number event. When $PE(X)$ we have $begin(X) < begin(take(X)) \leq end(take(X)) < end(X)$.
2. The domain of cs is the set of all terminating protocol execution events. When $PE(X)$ and $terminating(X)$ we have $begin(X) < begin(cs(X)) < end(cs(X)) < end(X)$.
3. The domain of $\Omega(X, j)$ is the set of all pairs X, j so that $PE(X)$, $terminating(X)$, and $P_i(X)$ for $i \neq j$. When $Z = \Omega(X, j)$ is defined, we have that $P_j(Z)$ and either $PE(Z)$ or else Z is the initial event in P_j.

D: If X and Y are protocol executions in P_i, and $X \prec Y$, then $terminating(X)$.

Fig. 2. Basic properties of the Take-A-Number algorithm.

is not reasonable. We said above that an execution by P_j that has ended extends an open permission which allows other processes to access their critical sections while P_j is still in its remainder phase. This permission cannot last forever of course, and the following execution by P_j must cancel it. Now if $Z \to X$ are two executions by P_j, where $Z = \Omega(Y, j)$ and X is the successor of Z, then $x = take(X)$ cancels the open permission of Z. So, if $\neg(y \to x)$, the cancelation effect of x has its impact on Y and it cannot be the case that Y gets its permission from Z.

In the following section, we redo the proof of the mutual exclusion but in an abstract setting, that is, in a setting that is not necessarily connected with the Take-A-Number algorithm or with any algorithm at all.

4. An abstract high-level proof of the mutual-exclusion property

We describe in this section a set of axioms in a first-order language (Figs. 2 and 3) and then prove that they entail the mutual-exclusion property. Of prime importance in these axioms is the unspecified relation $\prec$. At this stage, one may interpret $A \prec B$ simply as $A \to B$ (that is, $A \prec B$ iff $end(A) < begin(B)$). But there is a reason why we prefer to use here $\prec$ rather than $\to$. In the second part of this article, when we deal with the Ricart and Agrawala algorithm, we will interpret $\prec$ as the causal ordering rather than the temporal precedence relation $\to$. So, to allow the flexibility of applying the $\prec$ relation to very different situations we use an "uninterpreted" symbol for our abstract presentation. This will allow us to apply our axioms to the two different mutual-exclusion algorithms of this paper.

We first define the signature of the language in which the axioms are stated. This is a system execution signature (see Definition 2.1) which contains two sorts: *Event* and *Moment*. There are predicates and functions enumerated as follows.

1. We have the following unary predicates on the events.
   (a) PE for Protocol Execution. [We write $PE(X)$ in order to say that event X is a high-level protocol execution.]
   (b) $terminating$. [We write $terminating(X)$ to say that X is a terminating execution.]
   (c) For some N (number of processes) we have predicates $P_1, \ldots, P_N$. [We write $P_k(X)$ to say that event X "is by" process P_k.]
2. There are two binary relations: $\prec$ and "dominates" defined on the events. We write $a \preceq b$ as a shorthand for $a \prec b \vee a = b$. [The role of "dominates" will be explained below.] In addition we have the binary relation $<$ on *Moment*, and the resulting $\to$ which are present in every system execution signature.
3. There are partial functions defined on events X: $take(X)$, $cs(X)$, and $\Omega(X, i)$ (for every $i \in \{1, \ldots, N\}$). The domains of these functions are described below.

The "axioms" or high-level properties are grouped into two lists: the basic properties, described in Fig. 2, and the crucial properties described in Fig. 3. The basic properties are easy to prove for the Take-A-Number algorithm, and the more delicate crucial properties were proved at the end of the previous section.

Assume that X and Y are protocol executions, and let $x = take(X)$, $y = take(Y)$. Say $P_j(X)$, $P_i(Y)$ and $i \neq j$.

1. Either x dominates y, or y dominates x (but not both).
2. If $y \prec x$, then x dominates y.
3. If Y is terminating and $\Omega(Y, j) \prec X$, then $y \prec x$.
4. Assume that Y is terminating and $\Omega(Y, j) = X$. Then
   (a) $x \prec cs(Y)$, and
   (b) x dominates y, or X is terminating and $cs(X) \prec cs(Y)$.

Fig. 3. Crucial high-level properties of the Take-A-Number Bakery algorithm.

The mutual-exclusion property is the following statement.

For all X and Y, if $PE(X)$ and $PE(Y)$ and $X \neq Y$, if *terminating*(X) and *terminating*(Y), then $cs(X) \prec cs(Y)$ or $cs(Y) \prec cs(X)$.

We are interested in a somewhat stronger form in which not only the critical section events are linearly ordered, but their ordering is determined by the dominance relation of the take-a-number events.

Definition 4.1. The stronger mutual-exclusion property is the following property. Suppose that X and Y, are protocol executions in different processes, $x = take(X)$ and $y = take(Y)$. Suppose that *terminating*(Y) and *y dominates x*. Then *terminating*(X) and $cs(X) \prec cs(Y)$.

So, the stronger mutual exclusion does not say directly that the critical section events are never concurrent, but rather that their ordering is determined by the dominance relation. The following short argument shows that the mutual-exclusion property is indeed a consequence of the "stronger" mutual-exclusion property. Assume that X and Y are two terminating protocol executions, and we shall prove that $cs(X) \prec cs(Y)$ or $cs(Y) \prec cs(X)$. There are indices i and j such that $P_j(X)$ and $P_i(Y)$ (by basic property **A**). By **C1** $x = take(X)$ and $y = take(Y)$ are defined. Suppose first that $i = j$. Since $X \neq Y$, we either have $X \prec Y$ or $Y \prec X$ by **B**. Assume for example that $X \prec Y$. Then $end(X) \prec begin(Y)$ by **B**, and then **C2** implies that $end(cs(X)) \prec end(X)$ and $begin(Y) \prec begin(cs(Y))$. So $end(cs(X)) \prec begin(cs(Y))$ (equivalently $cs(X) \rightarrow cs(Y)$). By property **A**, these events are by the same process, and hence $cs(X) \prec cs(Y)$ by **B** as required.

Suppose next that $i \neq j$. Then by the first Crucial Property we have that either x dominates y or y dominates x. Now if y dominates x then the stronger mutual-exclusion property yields $cs(X) \prec cs(Y)$, and if x dominates y then $cs(Y) \prec cs(X)$.

Theorem 4.2. *Within the framework of Tarskian system executions the axioms of Figs. 2 and 3 imply the stronger mutual-exclusion property.*

Proof. We shall be very careful to rely in the proof only on properties listed in Figs. 2 and 3 and not on any intuition or additional properties that the Take-A-Number algorithm gives. Since we deal here only with Tarskian system executions, the theorem states that any system execution that satisfies the properties of Figs. 2 and 3 also satisfies the stronger mutual-exclusion property. Thus properties that hold in any system execution can surely be used in the proof, such as the transitivity of the $<$ relation on the *Moment* domain.

Suppose that X and Y are as in Definition 4.1. That is, they are protocol executions in different processes, Y is terminating, and for $x = take(X)$ and $y = take(Y)$ we assume that *y dominates x*. Let i and j be the indices of Y and X (respectively). By assumption, $i \neq j$. If $y \prec x$, then *x dominates y* by the second Crucial Property, which contradicts our assumption that *y dominates x*, since dominance is anti-symmetric by the first item. Thus we get

$$\neg(y \prec x). \tag{5}$$

As Y is terminating, $\Omega(Y, j)$ is defined, and is either a protocol execution in P_j or the initial event there (by **C3**). Since both X and $\Omega(Y, j)$ are in P_j, the seriality property implies that we have one of the following three possibilities.

Case 1: $\Omega(Y, j) \prec X$. Then Item 3 in the Crucial Properties list implies that $y \prec x$ which contradicts conclusion (5) above.

Case 2: $X \prec \Omega(Y, j)$. Say, $Z = \Omega(Y, j)$. By Crucial Property 4(a) we have $take(Z) \prec cs(Y)$. Since $X \prec Z$, we get that X is terminating (by **D**) and hence that $cs(X)$ is defined (by **C**(2)). We shall prove that $cs(X) \prec take(Z)$, and then $cs(X) \prec cs(Y)$ follows by transitivity of $\prec$ (item **B**), as required.

To prove $cs(X) \prec take(Z)$, we first get $end(X) < begin(Z)$ (since $X \prec Z$ and both events are in P_j). We also have (by item **C**) $end(cs(X)) < end(X)$, and $begin(Z) < begin(take(Z))$. Hence $cs(X) \rightarrow take(Z)$, and as these events are both in P_j we get $cs(X) \prec take(Z)$ (by **B** again).

Case 3: $X = \Omega(Y, j)$. Then item 4(b) implies that x dominates y, or X is terminating and $cs(X) \prec cs(Y)$. But it is not the case that x dominates y, and hence *terminating*(X) and $cs(X) \prec cs(Y)$ follow immediately. □

5. Modeling message passing

The main aim of this section is to described the causal ordering relation of Lamport [14] which will be used in Section 6. Although we use the standard definitions of message-passing systems (as defined for example in chapter 2 of [3]), we present them in the context of Tarskian system executions, and hence our detailed presentation.

Suppose two serial processes, say $Server_1$ and $Server_2$ ("sender" and "receiver"), that communicate by means of a channel $C_{1,2}$ which consists of two queues, one at the sender's side of the channel for "outgoing" messages that have been sent but have not yet arrived to the other side of the channel, and the other queue at the receiver's side for "incoming" messages that "crossed" the channel and are waiting to be delivered to $Server_2$. At any state S the channel's state $C^S_{1,2} = (outgoing^S, \ incoming^S)$ is a description of the sequences of messages at the two queues. (Although it is possible to unify these two queues into a single queue, and the system would be virtually the same, we slightly prefer this separation since it makes it very clear that a message sent is not immediately available to the receiving process.) Using states of the queues we define three types of steps concerned with $C_{i,j}$.

1. A "send" step (or event) by $Server_1$ sending message m on $C_{1,2}$ is represented by a pair of states (S, S') where $C^{S'}_{1,2}$ is obtained from $C^S_{1,2}$ by adding m at the tail of its outgoing message queue.
2. A "transfer" step is enabled at a state S when its outgoing queue is non-empty, and then the step is a pair (S, S') where $C^{S'}_{1,2}$ is obtained by removing message m from the head of the outgoing message queue and adding it to the tail of the incoming queue. (For simplicity we are considering order preserving channels, although this property will not be used.) We say that m is the value of this event.
3. A *remove-and-compute* step by $Server_2$ is enabled at state S when the incoming queue is non-empty. The receiving process removes a message from the head of that queue and is now ready to act upon it as instructed by its program. (Remove-and-compute steps have an additional role concerning the timestamps which is explained at the end of this section.)

In our application (the Ricart and Agrawala algorithm) we have N processes $Server_1, \ldots, Server_N$ with channels connecting each pair of distinct processes. A global state now describes all channels and the local states of each of the processes. A step is a pair of global states (S, S') so that one of the following is the case:

1. One of the processes executes a send (or a *remove-and-compute*) instruction as its program dictates, and both its local state and the state of the corresponding outgoing (or incoming) queue is changed. The local states of other processes and all other queues do not change. (*remove-and-compute* steps can be executed only on non-empty incoming queues.)
2. One of the channels executes a transfer step in which a message is transferred from the outgoing into the incoming queue. No other queue is changed and the local states of the processes remain the same in S and S'.
3. A local step is an execution of an instruction by one of the processes that does not involve the channels' queues. For example, a write on a private variable is a local step. Also, after removing a message, the process has to execute several local steps with this message as a parameter.

A *history* (also called run) is a sequence $R = (S_0, S_1, \ldots)$ of global states so that all queues are empty at S_0, all processes are at their initial state in S_0, and for every i (S_i, S_{i+1}) is a step as described above.

So when P_i executes a send instruction, the message to P_j is immediately added to the channel's outgoing queue, but the transfer of that message to the incoming queue and then the removal and action on that message by P_j may happen at arbitrary later times in the history. We assume however that any message is finally transferred, and no process will delay forever removal of a message from a non-empty incoming channel queue.

Standing alone, history R is not a Tarskian system execution, and we shall employ the method described in 2.1.1 to transform R into such a structure. We first define a system execution signature of the message-passing language (see Definition 2.1 for system execution signatures).

1. The sorts are *Event*, *Moment*, and *Data*. (The events represent the send and the receive events. *Moment* is interpreted with the set of natural numbers, and *Data* is for the set of all possible message values.)
2. In addition to the predicates and functions that are common to all system executions (Definition 2.1) we have predicates *send* and *receive* (which is synonymous with *remove-and-compute*) over the events, and predicates $C_{i,j}$ (that determine to which channel these events belong to). We also have predicates $P_1, \ldots P_N$ (to define the extension of each process) and two functions *Val* and γ whose usage is explained in the following definition and in the ensuing discussion.

   So for example $send(e)$ says that e is a send event, and $P_3(e)$ that process P_3 is the sending process, and $C_{3,5}(e)$ that the message is sent on the channel connecting P_3 to P_5.
3. The *Val* function is from *Event* to *Data*. So, for example, if $send(e)$ then $Val(e)$ is the value of the message of e.
4. The γ function is from the *receive* events into the *send* events.

Definition 5.1. A system execution that interprets this message-passing signature is said to be a "message-passing system execution" if the following hold.

1. All *send* and *receive* events are terminating. For every process P_i the events that belong to P_i are linearly ordered under $\rightarrow$. (Recall from Definition 2.1 that $\rightarrow$ is the temporal precedence relation.)

2. The *send* and the *receive* events related to any fixed channel $C_{i,j}$ are linearly ordered under $\rightarrow$. The set of events in one channel are disjoint to the set of events of any other channel.
3. For every *receive* event r, $\gamma(r)$ is a *send* event and $Val(r) = Val(\gamma(r))$. If $P_j(r)$ and $P_i(\gamma(r))$, then $C_{i,j}(r)$ and $C_{i,j}(\gamma(r))$. (This item expresses the assumption that the channel is reliable: the value of any message received is the value of the corresponding send event.)
4. For every *receive* event r, $\gamma(r) \rightarrow r$. (Intuitively: the sending of a message temporally precedes its reception.)
5. We say that a channel is order preserving when γ is order preserving on that channel. That is, whenever r_1 and r_2 are receive events on the channel and $r_1 \rightarrow r_2$, then $\gamma(r_1) \rightarrow \gamma(r_2)$. (Intuitively: messages are received in the order they were sent.)
6. No message duplication: γ is one-to-one.
7. γ is onto the set of *send* events. (The channel is lossless and the receiving process is attentive, so that each message sent finally arrives and is removed by the receiver process which proceeds with the required computations.)

We now explain (following 2.1.1) how a history $R = (S_0, S_1, \ldots)$ can be transformed into a message-passing system execution M. The set of low-level event of M is the set of all steps (S_i, S_{i+1}) in R that are *send*, *remove-and-compute*, or internal steps of the processes not connected with the channels. (Thus transfer events need not appear in M, as the executing processes are not aware of their existence.) The precedence relation is defined on these events: if $a = (S_i, S_{i+1})$ and $b = (S_j, S_{j+1})$ then $a \rightarrow b$ iff $i < j$. Functions and predicates are defined in the obvious way. For example, if $r = (S_i, S_{i+1})$ is a *remove-and-compute* step by P_i of message m in its incoming $C_{j,i}$ queue, then $r \in Event^M$, predicate $receive^M(r)$ holds, $Val^M(r) = m$, and the function $\gamma^M(r) = s$ is defined so that s is that step in R that corresponds to the sending of m. Additional predicates and functions may appear in M; for example details about the connection between the events and the programs that the processes execute.

An important feature is the introduction of high-level events into M (see 2.0.3). These are non-empty finite sets of step-events that represent operation executions. If A and B are such high-level events, then we define $A \rightarrow B$ if for every $a \in A$ and $b \in Ba \rightarrow b$ holds. In a similar fashion $A \rightarrow b$ and $a \rightarrow B$ can be defined when a and b are lower-level events.

The functions *begin* and *end* can be defined as follows on the events. If $s = (S_i, S_{i+1})$ is any step, then we can set $begin(s) = end(s) = i$. If A is a high-level event and $s = (S_i, S_{i+1})$ is the first step in A then $begin(A) = i$ and if $s = (S_j, S_{j+1})$ is its last step then we define $end(A) = j$.

Causality

The happened-before "causal" ordering was defined by Lamport [14], and since it is an important ingredient in the Ricart and Agrawala algorithm we report a modification of its definition (adapted to system executions) as follows. Let S be any Tarskian system execution that is a message-passing structure as defined above (in 5.1). The "causal" ordering $\prec_c$ is defined over the events in S as the transitive closure of the following relation:

1. If both e_1 and e_2 belong to the same process (that is $P_i(e_1)$ and $P_i(e_2)$) and if $e_1 \rightarrow e_2$, then $e_1 \prec_c e_2$.
2. If r is any *remove-and-compute* event, then $\gamma(r) \prec_c r$.

In a more explicit way we can say that $a \prec_c b$ holds iff there is a sequence of events $e_1, \ldots, e_k$ such that $e_1 = a, e_k = b$, and for every i either e_i and e_{i+1} belong to the same process and $e_i \rightarrow e_{i+1}$, or else e_i is a *send* event e_{i+1} a *remove-and-compute* event and $e_i = \gamma(e_{i+1})$.

It follows that if $a \prec_c b$ are by different processes then there has to be at least one index i with $e_i = \gamma(e_{i+1})$.

Observe that (by item 4 of Definition 5.1) $a \prec_c b$ implies that $a \rightarrow b$. Hence $\prec_c$ is irreflexive. Clearly $\prec_c$ is a transitive relation. For events that are in the same process, $\prec_c$ coincides with $\rightarrow$. But in general $a \rightarrow b$ does not imply $a \prec_c b$.

It is convenient (for notational reasons) to extend the causal relation over high-level events as well. If X and Y are high-level events then we define $X \prec_c Y$ if $x \prec_c y$ holds for all $x \in X$ and $y \in Y$. This definition is symmetric to the definition of $X \rightarrow Y$ iff $x \rightarrow y$ holds for all $x \in X$ and $y \in Y$. So we immediately have:

Lemma 5.2. *If $X \prec_c Y$ then $X \rightarrow Y$.*

5.1. Timestamp management

The Ricart and Agrawala algorithm described in Section 6 is an improvement of a mutual-exclusion algorithm of Lamport [14], and both algorithms employ the timestamping algorithm of Lamport described in [14] for extending the causal ordering $\prec_c$ into a linear ordering of the events. We describe this timestamp management algorithm next.

We assume that every process P_i has a local variable $timestamp_i$ which carries natural numbers and is initially 0. We assume that each message m has two fields: m.data is the data value of the message, and $m.ts$ is a natural number. When P_i executes an instruction "send message *val* to P_j" it actually does the following with a local variable m:

1. m.data := *val*;
2. $m.ts := timestamp_i$;
3. send m on the channel $C_{i,j}$ to $Server_j$.

from:	to:	responsible:
"Remainder"	"Requesting"	$User_i$
"Requesting"	"Waiting"	$Server_i$
"Waiting"	"CS"	$Server_i$
"CS"	"Exit"	$User_i$
"Exit"	"Remainder"	$Server_i$

Fig. 4. Phases of $Status_i$ and the process parts responsible for changes. The initial value of $Status_i$ is "Remainder".

Code for $User_i$

1. $Status_i$:= "Requesting";
2. **wait until**
   $Status_i$ = "CS";
3. critical section
4. $Status_i$:= "Exit";
5. **wait until**
   $Status_i$ = "Remainder".

Fig. 5. Ricart and Agrawala: Protocol for $User_i$. Initially, the value of $Status_i$ is "Remainder", and $User_i$ executes this protocol whenever it wishes to enter its critical section.

When P_i executes a *remove-and-compute* instruction of the form "with message m from P_j **do**...", it actually executes the following.

1. It dequeues the message m at the head of the incoming $C_{j,i}$ queue (assuming this queue is non-empty).
2. If $m.ts \geq timestamp_i$, then it sets $timestamp_i := m.ts + 1$, and
3. is ready to continue and to do with m.data what the algorithm instructs to do.

If e is an event by P_i then the value of $timestamp_i$ immediately after e is denoted $timestamp(e)$. If e is a *remove-and-compute* event by P_i then the value of $timestamp_i$ may increase with e, but for other events the value of $timestamp_i$ before and after e is the same. If s is a send event and r is the corresponding *remove-and-compute* event $s = \gamma(r)$, then $timestamp(s) < timestamp(r)$. The following lemma is a direct consequence of the definition of $\prec_c$.

Lemma 5.3. *If a and b are events by P_i and P_j where $i \neq j$, then $a \prec_c b$ implies that $timestamp(a) < timestamp(b)$.*

6. The Ricart and Agrawala algorithm

The Ricart and Agrawala algorithm [23] is a critical section algorithm that works for a message-passing communication medium, but is so similar in its spirit to the Bakery algorithm of Lamport that one can say that it is the message-passing version of that algorithm. In shared memory critical section algorithms, a process at an external non-critical section (remainder phase) is inactive from the point of view of the algorithm. The Ricart and Agrawala algorithm however requires that even when a process is at its remainder phase it continually answers request messages sent to it by other processes. We therefore find it convenient to separate each process P_i ($1 \leq i \leq N$) in two parts: $User_i$ and $Server_i$. $User_i$ is the process that requests critical section entries and which may also be in its remainder phase completely oblivious to the algorithm messages, while $Server_i$ is always actively monitoring the incoming channels and is sending request messages and responses.

These two parts of P_i share an atomic register $Status_i$ which both can read and write. No process other than P_i can access this local register. The values carried by $Status_i$ are the five status values: "Remainder", "Requesting", "Waiting", "CS", and "Exit", and "Remainder" is its initial value. In Fig. 4 we mark which of the two process parts is responsible for each change of $Status_i$. $User_i$ initiates the changes from "Remainder" to "Requesting", and from "CS" to "Exit". And $Server_i$ is responsible for the changes from "Requesting" to "Waiting", from "Waiting" to "CS", and from "Exit" back to "Remainder".

The protocol for $User_i$ is simple (Fig. 5). Whenever $User_i$ intends to enter its critical section it changes its $Status_i$ register from the "Remainder" to the "Requesting" value, and waits for its $Server_i$ process to change it to "CS". This signals to $User_i$ that it can enter its critical section, and upon exit from its critical section, $User_i$ sets $Status_i$ to the "Exit" value, and waits for $Server_i$ to change $Status_i$ to "Remainder". Only then $User_i$ returns to its remainder phase, and at any time thereafter it may execute this protocol again.

Every two server processes are connected with a bi-directional channel, assumed to be reliable, lossless, and without message duplication (order-preservation is not needed). Each message sent from $Server_i$ to $Server_j$, is delivered at some later time to $User_j$ which acts in accordance to its protocol. As explained in the previous section each message m has two fields: m.data is the data of the message, and $m.ts$ is a natural number (the timestamp value). For messages sent by P_i, the two values that m.data can take are the following:

1. "P_i requests entry", and
2. "OK".

Code for $Server_i$
repeat_forever

1. **if** $Status_i$ = "Requesting" **then**:
   (a) $ts_i := timestamp_i$;
   (b) send "P_i requests entry" to each P_j for $j \neq i$;
   (c) $Ack_i := 0$;
   (d) $Status_i$:= "Waiting"; $bag_i := \emptyset$.
2. **if** $Status_i$ is "Waiting" **then** with message m from P_j **do**:
   (a) **if** m.data = "OK" **then** $Ack_i := Ack_i + 1$;
   (b) **if** m.data = "P_j requests entry" **then**
   (**if** $(m.ts, j) < (ts_i, i)$ **then** send "OK" to P_j
   else put m in bag_i).
3. **if** $Status_i$ = "Waiting" **and** $Ack_i = N - 1$ **then** $Status_i$:= "CS".
4. **if** $Status_i$ = "Exit" **then**:
   (a) for every message m of value "P_j requests entry" in bag_i, remove m from bag_i and send "OK" to P_j;
   (b) $Status_i$ = "Remainder".
5. **if** $Status_i$ = "Remainder" **then** with message m of value "P_j requests entry" **do**:
   send "OK" to P_j.

Fig. 6. Ricart and Agrawala Protocol for $Server_i$. Local variables of $Server_i$ are $timestamp_i$ (initially 0), ts_i, Ack_i, and bag_i. $Status_i$ is shared with $User_i$.

The algorithm of $Server_i$ is the heart of the Ricart and Agrawala algorithm. Described in Fig. 6, it consists of five instructions, numbered 1 to 5, each of which is a conditional **if** statement that depends on the value of variable $Status_i$. We say that an instruction is enabled at a certain state if it can be executed at that state. For example, the instruction at line 2 is enabled when $Status_i$ = "Waiting" and there exists at least one non-empty incoming queue.

$Server_i$ is a serial process and it repeats forever the following routine: it reads register $Status_i$ and executes an instruction that corresponds to the value read. When an instruction of the form "with message m from P_j **do** …" is executed, there may be more than one non-empty incoming queue and in this case any non-empty incoming queue is arbitrarily chosen and the head message m is removed and executed on. (We require that no channel is left unattended for ever.)

It is clear from the codes of $User_i$ and $Server_i$ that the shared variable $Status_i$ cycles through the values "Remainder", "Requesting", "Waiting", "CS", and "Exit" (with "Remainder" as its initial value). So $Status_i$ cannot jump from "Remainder" to "CS" for example without passing through "Requesting" and "Waiting". Hence the code for $Server_i$ is executed in order: an execution of line 5 followed by executions of lines 1–4, and back to 5.

We recall the general timestamp management of Lamport described in Section 5. We assume that every $Server_i$ has a local variable $timestamp_i$ which carries natural numbers and is initially 0. When $Server_i$ executes an instruction "send message val to P_j" (in lines 1(b), 2(b), 4(a), and 5) it actually does the following with variable m:

1. m.data := val;
2. $m.ts := timestamp_i$;
3. send m on the channel to $Server_j$. (6)

When $Server_i$ executes an instruction "with message m from P_j **do**..." (in lines 2, 5), it actually executes the following. It removes (dequeues) message m from the head of the P_j to P_i incoming queue (assumed to be non-empty). If $m.ts \geq timestamp_i$, then it sets $timestamp_i := m.ts + 1$ (but otherwise $timestamp_i$ remains unchanged). Then it continues and does with m.data what the algorithm instructs to do.

We go over the instructions of the algorithm of $Server_i$ (Fig. 6) and make some definitions and explanatory notes. The process checks the value of register $Status_i$:

1. If $Status_i$ is "Requesting", then the value of $timestamp_i$ is first recorded in variable ts_i (line 1(a)). Then in line 1(b) $Server_i$ sends (in any order) to all the other servers the same message m where m.data is "P_i requests entry", and $m.ts = timestamp_i$. When all $N - 1$ messages are sent, the Ack_i variable of $Server_i$ is set to 0 and $Status_i$ is changed to "Waiting". In addition, variable bag_i (a set of messages) is set to the empty value $\emptyset$. The execution of line 1 (or any other line) is not assumed to be atomic. That is, it may be interleaved (or concurrently executed) with events by other processes. The execution of line 1(a) is called a take-a-number event.
2. While $Server_i$ is in the "Waiting" phase, it accepts incoming confirmation messages ("OK" messages) and at each such receive event it increases its local Ack_i variable. (It will also increase its $timestamp_i$ variable, as we explained above, if the incoming "OK" message m has $m.ts \geq timestamp_i$.)

   Since the value of $timestamp_i$ may increase (with incoming messages) it is important to notice that in line 2(b) the value of ts_i is as it was previously determined in line 1(a) rather than the actual value of $timestamp_i$.

While at the "Waiting" phase, $Server_i$ may also receive "P_j requests entry" messages, m, and it responds to these messages in two different ways in 2(b) (this distinction is the heart of the Ricart and Agrawala algorithm).

(a) If $m.ts$ is such that $(m.ts, j) < (ts_i, i)$ in the lexicographical ordering (where j is the index of the sender of message m), then an "OK" message is sent to $Server_j$.

(b) If however, $(m.ts, j) > (ts_i, i)$ (equality would not be possible since $i \neq j$), then the request m is added to bag_i.

3. If P_i remains in its "Waiting" phase forever, then this execution is defined to be non-terminating, but if Ack_i reaches the value $N - 1$, $Server_i$ sets the $Status_i$ variable to "CS", which enables $User_i$ an entry into its critical section. Assuming that any stay in the critical section is terminating, and assuming that the process never fails and is always active, then an execution in which Ack_i reaches the value $N - 1$ is terminating.

   While at "CS", $Server_i$ is inactive.
4. When $User_i$ leaves its critical section and enters the "Exit" phase, instruction 4 is enabled for $Server_i$. Executing this instruction, all messages are removed from bag_i and replied with "OK" messages. When done, $Server_i$ changes $Status_i$ to "Remainder". Now $User_i$ may continue its non-critical activities, and may return to a new "Requesting" phase in another cycle of execution.
5. If $Status_i =$ "Remainder" then $User_i$ immediately sends an "OK" message to P_j whenever it receives a "P_j requests entry" message. As explained above, the protocol does not show the underlined mechanism by which $m.ts$ is compared to the local variable $timestamp_i$ and then $timestamp_i := \max(timestamp_i, m.ts + 1)$ is executed.

A "protocol execution" X by P_i is a high-level event that contains the following lower-level events by both $Server_i$ and $User_i$:

1. X begins with the assignment of "Requesting" to $Status_i$ (instruction 1 in the code for $User_i$). But the first "interesting" event in X is the execution of line 1(a) ($ts_i := timestamp_i$) by $Server_i$. This "take-a-number" event is denoted $take(X)$.
2. X includes all the events that follow this first event by both $Server_i$ and $User_i$ as dictated by their protocols. If X contains an execution e of 4(b), then X terminates and e is its last event (assigning "Remainder" to $Status_i$).

There are two types of protocol executions: *non-terminating* and *terminating*. In case instruction 3 of $Server_i$ is never enabled, the execution never reaches the critical section and the corresponding execution is called *non-terminating*. Otherwise, the execution is a *terminating* execution. A *non-terminating* protocol execution thus includes executions of instructions of lines 1–2 that contain less than $N - 1$ executions of line 2(a) that increase the value of variable Ack_i.

Note that a *terminating* execution contains a critical section event and the execution of instruction 4 of the $Server_i$ protocol, but it does not contain the events pertaining to line 5 since it terminates with the execution of 4(b). It would be unreasonable to have the remainder events in the protocol execution since, from the point of view of the user, the execution has terminated with 4(b). Yet, although the "Remainder" phase that follows X is not included in X, it is conceptually connected with X (see also the footnote at page 2743).

If X is a *terminating* protocol execution then $cs(X)$ denotes the critical section event in X.

Assuming that all critical section events are terminating and that all processes are active and never fail, it follows that all protocol executions are *terminating*. This, however, is something that has to be proved and hence the existence of *non-terminating* protocol executions is not ruled out at this stage of our discussion. Although liveness properties are extremely important, we leave them aside in this paper since the mutual-exclusion property suffices for our aim of showing how the two algorithms can be compared by means of Tarskian system executions.

In order to highlight the resemblance of the Ricart and Agrawala algorithm with the Take-A-Number algorithm, we defined an execution of instruction 1(a) in a protocol execution X (by $Server_i$) as its "take-a-number" event, and notated it $take(X)$.

Definition 6.1. If $e = take(X)$, then the value of e, $Val(e)$, is the pair (n, i) where i is the index of the process P_i that executed this take-a-number event and n is the value of $timestamp_i$ assigned to variable ts_i in executing line 1(a) in e.

Definition 6.2. If e_1 and e_2 are take-a-number events with values (n_1, i_1) and (n_2, i_2) respectively and $(n_1, i_1) < (n_2, i_2)$ in the lexicographical ordering, then we say that e_2 dominates e_1.

It is obvious that if $i_1 \neq i_2$ then for any values of n_1 and n_2 either $(n_1, i_1) < (n_2, i_2)$ or $(n_2, i_2) < (n_1, i_1)$. So if e_1 and e_2 are take-a-number events by different processes then either e_2 dominates e_1 or else e_1 dominates e_2.

We say that protocol execution X is "normal" if each "OK" message received in an execution of 2(a) was sent (say by $Server_j$) as a response to the requesting message sent in X (and not in some earlier execution). We shall give a more formal definition of normality, and then outline a proof that all protocol executions are normal.

Consider the protocol of P_i as described in Figs. 5 and 6, and note the following property of its executions.

> For any protocol execution X, any message m removed from bag_i in an execution of 4(a) was previously introduced there by an execution of the **else** clause of 2(b) in X.

It follows from this and from similarly simple considerations of the code of $Server_i$ that any sending of message "OK" by process P_i to P_j done in executing 4(a) is preceded by a *remove-and-compute* event of the corresponding message "P_j requests entry" done in 2(b). Similar observations can be made for all other sending events of "OK" messages. There are three places

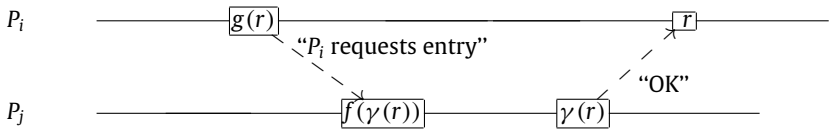

Fig. 7. Pictorial depiction of $g(r)$ where r is a receive event of an "OK" message by P_i, $u = \gamma(r)$ is the sending of that message, $t = f(u)$ is the receive event of the request from P_i, and $g(r) = \gamma(t)$ is the sending of that request.

in the code where "OK" messages are sent: in executions of 2(b), 4(a), and in 5 (in the "Waiting", "Exit", and "Remainder" phases). In each case, any sending of message "OK" by process P_i to P_j is preceded by a *remove-and-compute* event of the corresponding message "P_j requests entry". This observation enables the definition of a function f as follows. If s is any sending of message "OK" by process P_i to P_j, we let $f(s)$ be the (last) previous *remove-and-compute* event e by P_i of the corresponding message "P_j requests entry". The properties of f that will be used are that it is one-to-one, and also that if s is in protocol execution X then $f(s)$ is also in X, and if s is in the "Remainder" phase following X then $f(s)$ is also there.

Returning to the question of normality, we shall define now a function, g, which associates with every "OK" message received by P_i from P_j in a *remove-and-compute* event r a sending event $g(r)$ of a "P_i requests entry" message from P_i to P_j. The function f defined above and the function γ will serve in this definition, and the illustration of Fig. 7 will guide us.

Let X be a protocol execution by P_i, and r in X be a receive event that corresponds to an execution of instruction 2 (by $Server_i$) where the message obtained from P_j is of data value "OK". Consider $\gamma(r)$ which is the sending by P_j of the "OK" message received by r. We have defined $f(\gamma(r))$ as the corresponding receive by P_j of the message "P_i requests entry". Finally define $g(r) = \gamma(f(\gamma(r)))$ as the corresponding sending of that message. Since γ and f are one-to-one, g is also one-to-one. The definition of $\prec_c$ implies immediately that $g(r) \prec_c r$. We say that X is "normal" if $g(r) \in X$ for every $r \in X$ (a receive event of an "OK" message).

Lemma 6.3. *If X is any protocol execution by P_i, and r in X is a receive event that corresponds to an execution of instruction* 2 *where the message obtained from P_j is of data value "OK", then $g(r)$ is in X. That is, X is normal.*

If r_1, r_2 in X are two receive events of messages from P_{j_1} and P_{j_2} respectively of messages of data value "OK", then $j_1 \neq j_2$.

Proof. The protocol executions by P_i are linearly ordered and their order-type is either finite or that of the natural numbers. So if there is some X that is not normal, then there is a first one. Let r in X be a receive event of an "OK" message such that $g(r)$ is not in X. As $g(r)$ is a send event of a "P_i requests entry" message, it belongs to some execution Y of the protocol by P_i. Since $g(r) \prec_c r$ and $g(r)$ is not in X, $Y \to X$ follows (as $a \prec_c b$ implies $a \to b$ by Lemma 5.2). By minimality of X, Y is normal. As $Y \to X$, Y is *terminating*, and hence contains $N - 1$ receive events of "OK" messages, and for each such receive event v we have $g(v) \in Y$ by normality. Since Y (as any execution of the protocol) contains exactly $N - 1$ send events of "P_i requests entry" messages (executing line 1), and since the *terminating* Y contains $N - 1$ receive events of "OK" messages, it follows from the normality of Y that the one-to-one function g restricted to the receive events of "OK" messages in Y is onto the send events executed in line 1 in Y. Thus in particular $g(r)$ is of the form $g(v)$ for some v in Y, but as $r \neq v$ this is in contradiction to the fact that g is one-to-one.

The second part of the lemma is argued as follows. Suppose that $j = j_1 = j_2$ despite the assumption that $r_1 \neq r_2$. Since g is one-to-one and both $g(r_1)$ and $g(r_2)$ are in X (normality), we get in this case that there are two send events in X of "P_i requests entry" messages to process P_j. But this is not the case, because in line 1(b) of the protocol we see that the message is sent just once to each P_j for $j \neq i$. □

Since variable Ack_i is increased $N - 1$ times before $Status_i$ becomes "CS", if X is a *terminating* protocol execution by P_i then $N - 1$ messages were received which caused this increase in Ack_i. The second part of Lemma 6.3 shows that, in this case, these messages were received from different processes. Hence, if X is any *terminating* protocol execution by process P_i, then for every index $j \neq i$ there is a receive event of an "OK" message from P_j.

We can finally define the function $\Omega(Y, j)$ for every *terminating* protocol execution Y by P_i and process index $j \neq i$. Since Y is *terminating*, there are $N - 1$ executions of instruction 2 in Y that caused an increase of variable Ack_i, and (as we argued above) there is for every index $j \neq i$ a receive event r_j in Y of the "OK" message from P_j. Let $u = \gamma(r_j)$ be the corresponding send event. There are two possibilities: u can be in a protocol execution X by P_j, or else u can be in the "Remainder" phase of $Server_j$. In the first case we define $X = \Omega(Y, j)$ and in the second case we let $\Omega(Y, j)$ be the last protocol execution by P_j before u. (In case, u is in the initial "Remainder" phase, we let X be a hypothetical initial event by P_j which is not a protocol execution.)[3]

Lemma 6.4. *Let X be some protocol execution by P_i, and r in X be a remove-and-compute event of a "P_j requests entry" message m. Suppose that in executing instruction* 2(b) *for this message X finds that*

$$(m.ts, j) < (ts_i, i). \tag{7}$$

Let $s = \gamma(r)$ be the corresponding send event by P_j. Then there is a protocol execution Y by P_j such that $s \in Y$ and we have that $x = take(X)$ dominates $y = take(Y)$.

[3] In associating every "Remainder" phase R of $Server_j$ with the last protocol execution by P_j (if there is one) before R rather than with the one that follows R (if there is one) we are guided by the intuition gained with the Take-A-Number protocol. The "Remainder" phase corresponds to the write of value ∞ on R_j (line l_7).

Proof. By properties of the function γ (see Definition 5.1) $Val(s) = Val(r) = m$. Hence the data value of event s is "P_j requests entry". And as entry request messages are sent only by protocol executions (and not in "Remainder" phases), there is a protocol execution Y such that s is in Y (part of the execution of line 1). The required conclusion that x dominates y is a consequence of (7), as the following argument shows.

Recall (Section 5.1 and also Eq. (6)) that the timestamp field ts of any message that P_j sends is determined as the value of variable $timestamp_j$ at the time that the message is sent. When line 1 is executed in the "Requesting" phase, messages are sent but none is received, and hence the value of $timestamp_j$ does not change and it remains with its ts_j value. As m is a "P_j requests entry" message sent in Y and $y = take(Y)$, we get that $Val(y) = (m.ts, j)$. (See 6.1 for the definition of $Val(y)$.)

When P_i executes in X instruction 2(b), it uses the pair $(ts_i, i) = Val(x)$. So (7) implies that $Val(y) < Val(x)$, namely that x dominates y. □

High-level properties of the Ricart and Agrawala algorithm

We argue now that the abstract properties of Figs. 2 and 3, when suitably interpreted, hold in any execution of the Ricart and Agrawala algorithm.

Theorem 6.5. *The properties of Figs. 2 and 3 hold in any execution of the Ricart and Agrawala algorithm when $\prec$ is understood as $\prec_c$ (the causal ordering defined in Section 5).*

Proof. The basic properties (those of Fig. 2) are quite straightforward, and most of them were already established: such as the irreflexivity and transitivity of $\prec_c$. We shall only consider the Crucial Properties of Fig. 3.

Property 1 is obvious and was argued after Definition 6.2.

Property 2 follows Lemma 5.3.

For item 3, assume that Y is *terminating* and $\Omega(Y, j) \prec_c X$ where Y is by P_i and X is a protocol execution by P_j. By Lemma 5.2 we also have $\Omega(Y, j) \to X$. Let r be in Y the receive event of the "OK" message from P_j, and e be the sending of "P_i requests entry" from P_i to P_j in Y. By normality of Y (Lemma 6.3) we have that $e = g(r)$. In details this means that for $u = \gamma(r)$ and $t = f(u)$ we have that $e = \gamma(t)$. Event u is the corresponding sending of "OK" from P_j and t is the receive in P_j of the request of P_i. The definition of $\Omega(Y, j)$ implies that either $u \in \Omega(Y, j)$ or else u is in the "Remainder" phase of P_j that follows $\Omega(Y, j)$. In both cases $\Omega(Y, j) \to X$ implies that $t \to x$. This implies that $y \prec_c x$ as required for item 3 (by the chain $y \prec_c e \prec_c t \prec_c x$).

Now for item 4, suppose that $X = \Omega(Y, j)$. Using the same notation as in the previous paragraph, we have now that event t is in X or in the "Remainder" phase immediately following X. Since $x = take(X)$ is an execution of line 1(a), and t is either in an execution of lines 2 or 4 in X or else an execution of line 5 in the "Remainder" phase following X, $x \to t$. As $t \to u$ and $u = \gamma(r)$, $x \prec_c r$. As $r \to cs(Y)$, we get $x \prec_c cs(Y)$ as required by 4(a).

There are two possibilities for u, the sending of "OK" in X to P_i.

1. In case u is the delayed sending induced by the removal of the request of P_i from bag_j or u is in the "Remainder" phase immediately following X, then X is *terminating* and $cs(X) \prec_c cs(Y)$ follows (from $cs(X) \to u$, $u \prec_c r$, and $r \to cs(Y)$).
2. In case u is done for line 2(b) of the protocol, then condition $(m.ts, i) < (ts_j, j)$ implies that x dominates y (Lemma 6.4). □

Since Theorem 4.2 has only the axioms of Figs. 2 and 3 as assumptions, it follows immediately that the stronger mutual-exclusion property holds for the Ricart and Agrawala algorithm as well. Thus we have in correspondence with Theorem 3.4:

Theorem 6.6. *The Ricart and Agrawala algorithm satisfies the stronger mutual-exclusion property.*

7. Conclusion

The larger aim of our research is to indicate the possibility of using predicate language structures (Tarskian structures) to study concurrency in intuitive terms that enhance understanding of concurrent algorithms. We demonstrate this possibility here by investigating a new type of question in the area of distributed systems: the logical classification of algorithms in a manner that corresponds to pre-formal, intuitive similarities that one finds among algorithms in a family such as the Bakery algorithms. Specifically, we investigated here the Take-A-Number Bakery algorithm (a variant of Lamport's original [12]) and the Ricart and Agrawala algorithm [23], two algorithms that are seemingly very different (employing shared memory and respectively message passing) and yet are undeniably similar at the intuitive level. We explicate this intuition in a formal way by providing abstract properties that these two algorithms share, and by proving that the mutual-exclusion property is a consequence of these properties. We believe that a deeper understanding of these algorithms was obtained through our investigation, and we suggest that Tarskian event structures might be more appropriate for this new type of question than temporal logic structures which proved to be so efficient for other types of reasoning. The quest for common properties can be trivialized by the answer that the mutual-exclusion property itself is the common property, or perhaps by some other property that can be easily expressed, and hence, one may argue, there is no need for quantification languages and Tarskian structures for an answer. We believe however that a meaningful answer, namely one that clarifies the two algorithms and *explains* why they satisfy the strong mutual-exclusion property, requires a deeper analysis. We argued (see the discussion in Section 3.1) that the properties of Figs. 2 and 3 capture the essence of both algorithms at an abstract level that goes beyond the communication layer.

Our properties, or axioms, capture the essence of two different protocols, but we did not find those properties that are common to *all* Bakery like algorithms. That is, we did not answer the question "what is a Bakery algorithm?". The classical Bakery algorithms use unbounded natural numbers for timestamps, and there is an extensive effort by different authors both to find bounded valued timestamps and to modify the Bakery algorithm to a bounded value algorithm. (For example, see [11,25] and their lists of references). Our axioms clearly do not accommodate bounded valued Bakery algorithms since Item 2 of Fig. 3 says that

$$\text{If } y \prec x, \quad \text{then } x \textit{ dominates } y, \tag{8}$$

and this property implies that an infinite sequence of take-a-number events necessarily generates an infinite sequence of values. It would be interesting to find a more general set of properties that define the class of all Bakery algorithm—both bounded and unbounded, and [1] is possibly a step in this direction. In that work, Item 2 of Fig. 3 is replaced with

If $y \prec x$, then x *dominates* y, or the protocol event X is *terminating* and $cs(X) \prec end(y)$.

We conjecture that a similar property might be the clue to finding a list of properties that define the extension of all Bakery like algorithms, bounded and unbounded.

Another interesting question is to find a critical section algorithm for which it is not obviously clear whether it is or it is not a 'Bakery like' algorithm, and to use the method of this paper to provide a convincing answer.

Although a main aim of this research is to promote the usage of Tarskian structures in the study of concurrency, the global state approach with its history structures is a proven success and I believe that there is place for both the behavioral and the assertional approaches. Each is appropriate in some circumstances and less in some other. Exactly when and how is still mater of research and experience.

Acknowledgements

We are very fortunate, the article and its author, to have met such good friends, editors, and referees who improved its content and presentation. I wish to thank them all, but in name I will only mention R. Constable, L. Czaja, and M. Magidor.

References

[1] U. Abraham, Bakery Algorithms, in: L. Czaja (Ed.), Proc. of the CS& P'93 Workshop, Warszawa, 1994, pp. 7–40.
[2] U. Abraham, Models for Concurrency, Gordon and Breach, 1999.
[3] H. Attiya, N. Welch, Distributed Computing, Wiley, 2004.
[4] E. Börger, Y. Gurevich, D. Rosenzweig, The Bakery Algorithm: Yet Another Specification and Verification, Specification and Validation Methods, Springer, 1995, pp. 231–243.
[5] M. Botinčan, AsmL specification and verification of Lamports Bakery algorithm, Journal of Computing and Information Technology - CIT 13 4 (2005) 313–319.
[6] E.M. Clarke, The birth of model checking, in: O. Grumberg, H. Veith (Eds.), 25 Years of Model Checking: History, Achievements, Perspectives, in: Lecture Notes in Computer Science, vol. 5000, Springer-Verlag, Berlin, Heidelberg, 2008, pp. 1–26.
[7] R. Constable, M. Bickford, Formal Foundations of Computer Security, in: NATO Science for Peace and Security D: Information and Communication Security, vol. 14, 2008, pp. 29–52.
[8] Y. Gurevich, D. Rosenzweig, Partially ordered runs: a case study, in: Y. Gurevich, P.W. Kutter, M. Odersky, L. Thiele (Eds.), Proceedings of the international Workshop on Abstract State Machines, Theory and Applications (March 19–24, 2000), in: Lecture Notes in Computer Science, vol. 1912, Springer-Verlag, London, 2000, pp. 131–150.
[9] J.Y. Halpern, L.D. Zuck, A little knowledge goes a long way: knowledge-based derivations and correctness proofs for a family of protocols, Journal of the ACM 39 3 (July) (1992) 449–478.
[10] M. Herlihy, J. Wing, Linearizability: a correctness condition for concurrent objects, Transactions on Programming Languages and Systems 12 (3) (1990) 463–492.
[11] P. Jayanti, K. Tan, G. Friedland, A. Katz, Bounding Lamport's Bakery Algorithm, in: Proc. of the 28th Conf. on Current Trends in Theory and Practice of Informatics, in: Lecture Notes in Computer Science, vol. 2234, Springer-Verlag, 2001, pp. 261–270.
[12] L. Lamport, A new solution of Dijkstra's concurrent programming problem, Comm. ACM 17 (8) (1974) 453–455.
[13] L. Lamport, The synchronization of independent processes, Acta Informatica 7 (1) (1976) 15–34.
[14] L. Lamport, Time, clocks, and the ordering of events in distributed systems, Comm. ACM 21 (1978) 558–565.
[15] L. Lamport, A new approach to proving the correctness of multiprocess programs, ACM TOPLAS 1 (1) (1979) 84–97.
[16] L. Lamport, Reasoning about nonatomic operations, in: Proc. of the Tenth ACM Symposium on Principles of Programming Languages, ACM SIGACT-SIGPLAN, 1983, pp. 28–37.
[17] L. Lamport, On interprocess communication–part I: basic formalism, part II: algorithms, Distributed Computing 1 (2) (1986) 77–101.
[18] Z. Manna, A. Pnueli, Temporal Verification of Reactive Systems: Safety, Springer-Verlag, 1995.
[19] S. Merz, On the logic of TLA+, Computing and Informatics 20 (2001).
[20] A. Pnueli, The temporal logic of programs, in: Proc. of the 18th Annual Symposium on the Foundations of Computer Science, IEEE, November 1977, pp. 46–57.
[21] V. Pratt, Modeling concurrency with partial orders, International Journal of Parallel Programming 15 (1) (1986) 33–71.
[22] M. Raynal, A simple taxonomy for distributed mutual exclusion algorithms, SIGOPS Operating System Review 25 (2) (1991) 47–50.
[23] G. Ricart, A.K. Agrawala, An optimal algorithm for mutual exclusion, Comm. ACM 24 (1981) 9–17; Communications of the ACM 24, 578 (Corrigendum).
[24] M. Takamura, Y. Igarashi, Simple mutual exclusion algorithms based on bounded tickets on the asynchronous shared memory model, in: O.H. Ibarra, L. Zhang (Eds.), Proceedings of the 8th Annual international Conference on Computing and Combinatorics, August 15–17, 2002, in: Lecture Notes in Computer Science, vol. 2387, Springer-Verlag, London, 2002, pp. 259–268.
[25] G. Taubenfeld, The black-white bakery algorithm, in: Proc. of the 18th International Symposium on Distributed Computing, 2004, in: LNCS, vol. 3274, Springer Verlag, 2004, pp. 56–70.