Partially Ordered Runs: a Case Study

Yuri Gurevich[1] and Dean Rosenzweig[2]

Microsoft Research, USA
gurevich@microsoft.com and University of Zagreb, Croatia
dean@math.hr

Abstract. We look at some sources of insecurity and difficulty in reasoning about partially ordered runs of distributed ASMs, and propose some techniques to facilitate such reasoning. As a case study, we prove in detail correctness and deadlock–freedom for general partially ordered runs of distributed ASM models of Lamport's Bakery Algorithm.

Introduction

Distributed ASMs [Gur95] is a general concurrent model of multi–agent computation. It was intended, in generalization of its more limited precursors [GM90,GR93], to allow as much concurrency as logically possible. Although the definition has been in print for several years, its notion of partially ordered runs has remained largely unexploited in its generality—most of its uses in the literature have recoursed to some kind of specialization to linear time, discrete or continuous. The general partially ordered runs seem to be somehow difficult to handle and to reason about [1].

Apart from deeply engrained intuitions of linear time, we feel that some rather technical sources of this difficulty can be detected.

Sequential runs [Gur95] have two properties which greatly facilitate reasoning.

1. Every move is executed in a well–defined state.
2. 'External' (or 'monitored', cf. below) changes can be located in time, and thought of as actions by the environment. The environment thus becomes just another (typically implicit) agent, whose behaviour can be specified in declarative terms. The judicious splitting of dynamics to a part given by the program and a part that can be specified declaratively is a natural way to separate concerns and to abstract in ASMs.

In a partially ordered run neither of the above properties hold in general—a (global) state in which a move is executed is in general not uniquely defined, and it is not at all clear how to locate external changes in a partial order. These

[1] This remark is not limited to the ASM context—most formal methods modelling concurrency tend to fall back, one way or another, to some kind of interleaving, sequential semantics

seem to be important sources of insecurity and difficulty in reasoning about partially ordered runs.

We address both issues here, by developing some techniques to salvage as much of properties 1. and 2. as needed for partially ordered runs. In order to convince the reader that, with such means, nontrivial reasoning about partially ordered runs is feasible, we do it: as a case study, we apply the techniques to a nontrivial correctness proof. The case study also demonstrates that the proper setting for analysis of concurrent algorithms involves truly concurrent runs—mapping to linear time may well miss some important points.

In section 2 we deduce some simple consequences of the coherence condition of [Gur95], providing sufficient conditions for moves in a distributed run to have at least a significant portion of state well–defined when they execute. This largely reconstructs property 1. for partially ordered runs of many distributed programs.

In the same section we introduce 'external change' in form of 'monitored' moves, by unknown agents with unknown programs, located exactly in the partial order. This is an extension of the standard practice (of having 'the environment' as a single, typically implicit, unknown agent) reconstructing largely property 2. for partially ordered runs.

In the rest of the paper we explain a nontrivial correctness proof for partially ordered runs: we prove in detail correctness and deadlock–freedom of (distributed ASM models of) Lamport's Bakery Algorithm [Lam74]. We proceed on three different abstraction levels there. The abstraction level of our primary model, $\mathcal{B}_1$, corresponds precisely to that of Lamport's algorithm—the part specified declaratively is exactly what Lamport's algorithm doesn't define. A higher level description, of the model $\mathcal{B}_2$, allows us to deduce correctness and deadlock–freedom from abstract properties, shown to hold of $\mathcal{B}_1$. A lower level description, of the model $\mathcal{B}_0$, is an ASM shown to implement programatically exactly all behaviours allowed by $\mathcal{B}_1$.

Egon Börger and we wrote about the Bakery Algoirithm earlier; see [BGR95] where a correctness proof was given for ASM models of the Bakery Algorithm with runs embedded in continuous linear time. Our new models and proofs are similar to those of [BGR95], but they are also different. The proofs of [BGR95] rely essentially on continuous linear time. As we will see later (see section 7), certain information about partially ordered runs is obfuscated in linear runs. Here we remove the linear time crutches and work directly with partially ordered runs. As in [BGR95], we borrow ideas from [Lam74] and [Abr93]. In order for the paper to be reasonably self–contained we spell the entire construction out in full.

1 Preliminaries

We presume that the reader is familiar with [Gur95]. Consider a one-agent program π and let f be a basic function of π which is dynamic so that the values of f can change in runs of π. Egon Börger suggested to use for ASMs the following terminology borrowed from Parnas. f is *controlled* if only π can change it. f is

monitored if only the environment can change it. f is *shared* if both π and the environment can change it. (In [Gur95], controlled functions were called internal, and monitored functions were called external.)

The terminology extends naturally to a multi-agent program Π. Let X be a set of agents of Π. A dynamic basic function f is controlled by X if only agents in X can change it. f is monitored by X if none of the agents in X can change it. The terminology also extends to particular locations rather than whole functions.

2 Partially Ordered Runs

We rely on the notion of partially ordered run of [Gur95].

This means that we shall consider partially ordered sets of moves with the 'finite history' property: $\{y : y < x\}$ is finite for all x (we shall refer to the ordering relation by $<$, using also $\leq, >, \geq$). Each move is performed by an agent and, since agents are sequential, moves by one agent form a sequence; since there are finitely many agents, all antichains are finite.

A (global) state $\sigma(I)$ is associated with every finite initial segment I (a downwards closed finite subset) of a run, resulting from performing all moves in I so that if $s < t$ then s is executed earlier than t. In particular, if I is the empty segment then $\sigma(I)$ is the initial state of the run.

Using a partial order implies that moves s, t may be concurrent, i.e. incomparable: neither $s \leq t$ nor $t \leq s$ holds. The global state 'resulting' from a move is then in general not uniquely determined. It depends on what global state we see as the one in which the move is performed. Thus states are subject to a coherence condition [Gur95], which allows us to give the following definitions.

Let $\mathrm{Post}(t)$ be the set of all finite initial segments in which a move t is maximal, and let a be the agent performing t. For each $I \in \mathrm{Post}(t)$, the state $\sigma(I)$ is the state obtained when a executes its program in the state $\sigma(I \setminus \{t\})$

'The global state' in which a move is performed is also not in general uniquely defined. Let $\mathrm{Pre}(t) = \{I \setminus \{t\} : I \in \mathrm{Post}(t)\}$. Several different states are thus in general associated with $\mathrm{Pre}(t)$, which makes reasoning about partially ordered runs somewhat difficult. The coherence condition may however, as we shall see below, impose that some term has a unique value at all states $I \in \mathrm{Pre}(t)$.

In order to express such requirements succinctly, we extend term valuation from states to some statesets, saying that

$$\mathrm{Val}_{\mathrm{Pre}(t)}(u) = \begin{cases} c & \text{if } \forall I \in \mathrm{Pre}(t)\ (c = \mathrm{Val}_{\sigma(I)}(u)) \\ \text{undef} & \text{if no such value exists} \end{cases}$$

where t is a move in a distributed run, $\mathrm{Val}_S(u)$ is the value of term u in state S, [Gur95]. We shall often shorten $\mathrm{Val}_{\mathrm{Pre}(t)}(u)$ to $u_{\mathrm{Pre}(t)}$. When the value of $u_{\mathrm{Pre}(t)}$ is given by the first clause, i.e. when there is a c such that $\forall I \in \mathrm{Pre}(t)\ (c = \mathrm{Val}_{\sigma(I)}(u))$, we shall say that $u_{\mathrm{Pre}(t)}$ is *indisputable* (or that its value is indisputable). Notice that an indisputable value may also be undef, but whenever $u_{\mathrm{Pre}(t)} \neq \text{undef}$ then its value is indisputable.

For future reference, let us note some immediate properties of $\mathrm{Pre}(t)$.

Fact 1. *Let t be a move in a partially ordered run. The set $Pre(t)$ has the following properties.*

1. *$Pre(t)$ has a minimal element $\min Pre(t) = \{s : s < t\}$ and a maximal element $\max Pre(t) = \{s : s \not\geq t\}$.*
2. *$Pre(t)$ is the set of all initial segments I such that $\min Pre(t) \subseteq I \subseteq \max Pre(t)$, and is hence closed under unions and intersections.*
3. *Let s be another move. The following are equivalent:*
   (a) *s is concurrent with t.*
   (b) *$\min Pre(s) \cup \min Pre(t)$ is an initial segment that belongs to $Pre(s) \cap Pre(t)$.*
   (c) *$Pre(s) \cap Pre(t) \neq \emptyset$.*
   (d) *There exist $I, J \in Pre(t)$ with $s \in I \setminus J$.*

Statement 3 may need an argument. We prove that (a) implies (b) implies (c) implies (d) implies (a).

(a) implies (b). Assume that s is concurrent with t and let $I = \min \mathrm{Pre}(s) \cup \min \mathrm{Pre}(t)$. Clearly I is an initial segment. Check that $\min \mathrm{Pre}(t) \subseteq I \subseteq \max \mathrm{Pre}(t)$, so that $I \in \mathrm{Pre}(t)$. By symmetry, $I \in \mathrm{Pre}(s)$.

(b) implies (c). Trivial.

(c) implies (d). Assume that $\mathrm{Pre}(s) \cap \mathrm{Pre}(t) \neq \emptyset$ and let J be any member of $\mathrm{Pre}(s) \cap \mathrm{Pre}(t)$. Set $I = J \cup \{s\}$. Clearly, $I \in \mathrm{Pre}(t)$.

(d) implies (a). Suppose that $I, J \in \mathrm{Pre}(t)$ and $s \in I \setminus J$. If $s < t$ then $s \in \min \mathrm{Pre}(t) \subseteq J$ so that $s \in J$ which is impossible. The dual argument shows that $s > t$ is impossible as well.

We shall say that a move t (in a given partially ordered run) *may change* the value of term u if, for some $I \in \mathrm{Pre}(t)$, we have $\mathrm{Val}_{\sigma(I)}(u) \neq \mathrm{Val}_{\sigma(I \cup \{t\})}(u)$ (equivalently, if t changes the value of u in some linearization of the run). If the above holds for all $I \in \mathrm{Pre}(t)$ (equivalently, if t changes the value of u in all linearizations), we shall say that t *must change* the value of u.

Recall that a linearization of a partially ordered run is a run with the same moves, and a linear order extending the given partial order. It was noted in [Gur95] that, in view of the coherence condition, all linearizations of a finite run have the same final state.

Example 1. Take for instance two agents a, b, such that a executes the program

```
x := 1
```

and b executes the program

```
if mode = first then
   mode := second
   y := 1
endif
if mode = second then
   y := max(x,y)+1
   mode := final
endif
```

Now assume that $x = y = 0$, mode = first initially and consider a run with a move s of a, concurrent with two consecutive moves t_1, t_2 of b. Then both s and t_1 may but not must change the value of $\max(x, y)$, while t_2 must change it.

In these terms, we have

Lemma 1. *If $u_{Pre(t)}$ is not indisputable, then there is a move s concurrent with t which may change u.*

Proof. Assume the conclusion is false, that no move concurent with t may change u. To go from $\sigma(\min \mathrm{Pre}(t))$ to $\sigma(I)$, by Fact 1, we have to execute only some moves concurrent to t, none of which may change u. Thus $\mathrm{Val}_{\sigma(I)}(u) = \mathrm{Val}_{\sigma(\min \mathrm{Pre}(t))}(u)$ for all $I \in \mathrm{Pre}(t)$, and $u_{\mathrm{Pre}(t)}$ is indisputable, in contradiction to the premise. □

Lemma 2. *If there is move s concurrent with t which must change u, then $u_{Pre(t)}$ is not indisputable.*

Proof. Since s, t are concurrent, by Fact 1 there is an initial segment $I \in \mathrm{Pre}(s) \cap \mathrm{Pre}(t)$. Then both I and $I \cup \{s\}$ are in $\mathrm{Pre}(t)$, and, since s must change u, they have different values of u. □

We shall say that a term u is *focused* (in a given run) if any move which may change its value, also must change it. For a focused term u it is unambiguous to say that a move changes the value of u. In many cases the property of being focused will be obvious from the programs. Here we will also use the following lemma.

Lemma 3. *If the value of a term may only be changed by a single agent, then the term is focused.*

Proof. Suppose that u may be changed only by agent a and that t is a move by a. It suffices to show that $u_{\mathrm{Pre}(t)}$ is indisputable, since then $u_{\mathrm{Post}(t)}$ is also indisputable, and is different from $u_{\mathrm{Pre}(t)}$ iff t changes u. To see that $u_{\mathrm{Pre}(t)}$ is indisputable, note that the agents involved in all moves concurrent to t are different from a, and by the premise none of them may change u. Then by lemma 1 $u_{\mathrm{Pre}(t)}$ is indisputable. □

Putting the above lemmata together, we have

Fact 2. *If term u is focused, then $u_{Pre(t)}$ is indisputable iff t compares, in the given partial order, with all moves changing the value of u.*

The above example shows that the assumption of focus cannot be dropped for one direction of fact 2: $\max(x, y)_{\mathrm{Pre}(t_2)}$ is indisputably 1, and t_2 is concurrent to s, which may change the value. The nice property that every location is changed by one agent only does not help with the term $\max(x, y)$. Note also that fact 2 is useful for 'counter' updates like $c := c + 1$; if c is changed only in such a way, even concurrently, then it is focused.

Fact 3. *If the values of $u_{Pre(t)}$ and $u_{Pre(s)}$ are both indisputable but different, then $s < t$ or $t < s$.*

Proof. Assume $u_{\mathrm{Pre}(t)}$ and $u_{\mathrm{Pre}(s)}$ are both indisputable, and s and t are concurrent. Then there is an initial segment $I \in \mathrm{Pre}(s) \cap \mathrm{Pre}(t)$, and $u_{\mathrm{Pre}(s)} = \mathrm{Val}_{\sigma(I)}(u) = u_{\mathrm{Pre}(t)}$. □

A novelty of this paper is in allowing a run of a multi-agent program Π to have 'monitored moves', that is moves of unknown agents. We have some known agents with known programs. The moves of these agents are by definition *controlled*.

In addition, there may be some number of unknown agents whose programs are unknown as well. Their moves are by definition *monitored*.

A dynamic function of Π, or a location, will be called controlled if it is controlled by the known agents.

It will be called monitored if it is monitored by the known agents.

The presence of unknown agents is consistent with [Gur95], even though the standard practice has been so far to assume that all explicit moves belong to known agents with known programs, though the active environment could make some implicit moves.

The moves by the environment now become explicit, and the unique monitored agent of standard practice, 'the environment', is now allowed to split to a number of possibly different, unknown agents.

The total number of agents involved in any given state is still finite. The coherence condition applies to all moves, controlled or monitored (even though we may have no direct way to verify instances of the coherence condition that involve monitored moves). Therefore facts 2 and 3 remain valid.

The presence of monitored moves allows us to separate concerns and to abstract. Parts of the algorithm can be formulated in form of more or less declarative statements about monitored moves in runs (which blurs to an extent the distinction between the algorithm and its environment).

3 Lamport's Algorithm

For arbitrary but fixed N let $P_1, \ldots, P_N$ be processes (we shall also talk about 'customers') that may want from time to time to access a 'critical section' CS of code. Any mutual exclusion protocol—which each P_i is supposed to execute in order to enter the critical section—has to prevent two processes from being in the critical section simultaneously. The Bakery Algorithm provides each P_i with a (shared) register R_i and a (private) array $n[1], \ldots, n[N]$ holding natural numbers. Only P_i is allowed to write to R_i but every process can read the register. We assume each register to be initialized with value 0.

The algorithm was presented by Lamport with the following piece of pseudocode.

Start

```
n[i] := 1
write(R_i,n[i])
```

Doorway

```
for all j≠i, read(R_j,n[j])
```

Ticket

```
n[i] := 1 + max_j n[j]
write(R_i,n[i])
```

Wait

```
for all j≠i, repeat
   read(R_j,n[j]) until
   n[j]=0 or n[j]>n[i] or (n[j]=n[i] and j>i)
```

Critical Section

Finale

```
R_i := 0
```

The Bakery Algorithm is divided into six consecutive phases: *start*, *doorway*, *ticket* assignment, *wait* section, *critical section* and *finale*.

To declare its interest in accessing the critical session, a process P_i writes 1 into array variable n_i and then posts the written value in its register.

In the doorway section, P_i copies all the other registers into its array. It then computes a *ticket*, which is the least integer greater than all integers in its private array, writes the ticket into n_i and posts the written value in its register.

During the subsequent wait section, process P_i keeps reading, into its array, the registers of each other process P_j, until the resulting array value $n[j] = 0$ or $n[j] > n[i]$ or $n[j] = n[i] \wedge j > i$.

The meaning of the condition is the following: if $n[j] = 0$, then P_j is not interested in entering the critical section, and it has no right to block P_i. If $n[j] > n[i] > 0$, then P_i has a smaller 'ticket' and has the right to go before P_j. The last clause resolves the case of two customers obtaining the same 'ticket': then one with smaller identifier goes first. Note that by ordering pairs of positive integers lexicographically:

$$(i, j) < (k, l) \longleftrightarrow [i < k \text{ or } (i = k \text{ and } j < l)]$$

one can write the until condition as follows: n[j]=0 or (n[j],j)>(n[i],i).

Once permitted to go, P_i enters the critical section. Upon leaving CS, as finale, P_i sets its register to 0.

Note also that the for-all commands in the doorway and the wait section may be executed in many ways, in various sequences, all at once, concurrently etc.

It may be worth mentioning the following. The process first writes into $n[i]$ and then posts the written value at R_i. Obviously it could do the two actions in the reverse order. Intuitively, the order between the two actions is immaterial, but the sequential character of the pseudo-code imposes one.

4 The Primary Model: ASM $\mathcal{B}_1$

The doorway section in Lamport's program does not give us any indication how customer i is supposed to perform reading. Should it read the registers R_j in the order given by the indices, in the reversed order? Should it get help and use vassal agents, one per each R_j ? There are many other possibilities. To reflect the situation in proper generality, our primary ASM model $\mathcal{B}_1$ includes no reading instructions whatsoever. Instead, we will require that runs of $\mathcal{B}_1$ satisfy certain provisos that guarantee that reading is performed.

4.1 The Program of $\mathcal{B}_1$

The ASM has only one program, used by all customers, which has five rules. The array $A(X, Y)$ represents the array $n[Y]$ of the program, private to customer X. We assume that initially all registers have value 0, all customers are in mode satisfied, and all elements of the array $A(X, Y)$ are undef. We assume that the identifiers of the N customers are distinct natural numbers $< N$. Variables X, Y will range over customers.

Start

```
if mode(me) = satisfied then
   A(me,me) := 1, R(me) := 1, mode(me) := doorway
```

Ticket

```
if mode(me) = doorway and ∀Y ≠ me (A(me,Y) ≠ undef) then
   A(me,me) := 1 + max_Y A(me,Y), R(me) := 1 + max_Y A(me,Y)
   mode(me) := wait
```

Entry

```
if mode(me) = wait and
∀ Y ≠ me (A(me,Y)=0 or (A(me,Y),id(Y)) > (A(me,me),id(me))) then
   mode(me) := CS
```

Exit

```
if mode(me) = CS then
   mode(me) := done
```

Finale

```
if mode(me) = done then
   R(me) := 0, mode(me) := satisfied
   ∀ Y ≠ me A(me,Y) := undef
```

4.2 Semantics of $\mathcal{B}_1$

We would like to assume that, in any mode different from *Satisfied*, no customer stalls forever; eventually it makes a move (provided a move is continuously enabled from some time on).

Since in ASMs we have no explicit notion of a move (or program) being enabled, and in partially ordered runs we have no explicit notion of time, both 'enabled' and 'continuously from some time on' need definitions.

There are two obvious candidates for the notion of a program being enabled in a state. One is based on the intuition that a program is enabled if it 'gets down to updates', i.e. if in the given state it generates a nonempty set of updates. The other possibility is that it really changes the state, i.e. that the updateset is nonempty and also nontrivial. We are happy to sidestep the issue here, since for all programs of this paper the two notions will coincide—whenever a nonemtpy set of updates is generated, it will also be nontrivial. Thus we can say that a program is enabled in state σ if it produces a nonempty set of updates in σ.

We say that an agent X stalls forever in a run if (a) X has a last move, say t, and (b) after t a move by X (the program of X) is eventually always enabled (in all $\sigma(J)$ for $J \supseteq I$, for some initial segment $I \ni t$).

We thus assume

Progress Proviso. No customer, in mode other than *Satisfied*, stalls forever.

We consider the runs of $\mathcal{B}_1$ containing enabled moves by customers executing their programs, subject to the Progress Proviso, and also some monitored moves. Our entire knowledge about monitored moves will be encapsulated in explicit requirements D, W1, W2 below.

We now define intervals characterized by the successive executions, by a process X, of its rules Start, Ticket, Entry, Exit, Finale (also in a partial order we refer to open intervals $(a, b) = \{x : a < x < b\}$).

Definition 1. *Suppose a is the move of X executing* Start *rule, and b is the next move by X (which has to execute the* Ticket *rule).*

Then the interval $x = (a, b)$ is a doorway *of X, and $a = \mathsf{Start}(x), b = \mathsf{Ticket}(x)$. If b is the last execution of X then the* wait interval $W(x) = \{t : t > b\}$ *is incomplete and the* CS interval $CS(x)$ *is undefined. Suppose that c is the next move of X after b (necessarily executing* Entry *rule), d is the next move of X after c (necessarily executing* Exit *rule), and e is the next move of X after d (necessarily executing* Finale *rule). Then $W(x) = (b, c)$ and $CS(x) = (c, d)$, $c = \mathsf{Entry}(x)$, $d = \mathsf{Exit}(x)$, $e = \mathsf{Finale}(x)$.*

By Progress Proviso and requirement D below, every doorway is complete, i.e. each execution of Start is followed by execution of Ticket. So is every critical section, i.e. each execution of Entry is followed by executions of Exit (and subsequently Finale).

The program of customer X writes to locations $\mathrm{mode}(X), R(X), A(X,Y)$[2], where locations $A(X,Y)$ with $Y \neq X$ are only cleaned up (that is set to undef) by X (in Finale) and somebody else writes more meaningful information into these locations.

Our program covers all but the reading actions. Since our definitions do not allow 'partially known programs', i.e. a controlled agent can do no more than what his program instructs him to do, more meaningful values have to be written there by the environment, i.e. by somebody else.

We assume that locations $\mathrm{mode}(X), R(X), A(X,X)$ are also controlled by X, i.e. that no other (known or unknown) agent may write there. This is justified by Lamport's algorithm: other customers, as well as the environment, have no business writing to $\mathrm{mode}(X), R(X), A(X,X)$.

This assumption implies that $R(Y)$ is focused, for all customers Y, and, by the program and fact 2, for all moves t in a run of $\mathcal{B}_1$,

Corollary 1. *$R(Y)_{Pre(t)}$ is indisputable iff t compares to all* Start, Ticket *and* Finale *moves of Y.*

To avoid repetitive case distinctions for customers which (being satisfied) have register 0, and of customers which happen to receive the same ticket, we introduce the following notation. If f is a function from customers to natural numbers, let

$$f'(X) = \begin{cases} N \cdot f(X) + \mathrm{id}(X), & \text{if } f(X) > 0; \\ \infty, & \text{otherwise.} \end{cases}$$

Let X, Y range over customers, and x, y over doorways of customers X, Y respectively.

We abbreviate $1 + \max_Y A(X,Y)_{\mathrm{Pre}(\mathsf{Ticket}(x))}$ as $T(x)$.

The declarative requirements, saying what reads need be done, are then (with x being an arbitrary doorway of customer X)

D Each execution of Start by X completes to a doorway x. For each x, for each $Y \neq X$ there is a move $b \in x$, such that $A(X,Y)_{\mathrm{Pre}(\mathsf{Ticket}(x))} = R(Y)_{\mathrm{Pre}(b)}$ (thus $T(x) > R(Y)_{\mathrm{Pre}(b)} \neq$ undef).

W1 If $W(x)$ is complete, then for each $Y \neq X$ there is a move $b \in W(x)$, such that $R(Y)_{\mathrm{Pre}(b)} = A(X,Y)_{\mathrm{Pre}(\mathsf{Entry(x)})}$ (thus $T'(x) < R'(Y)_{\mathrm{Pre}(b)} \neq$ undef).

W2 If $W(x)$ is incomplete, then for some $Y \neq X$ there is an infinite chain $b_1 < b_2 < \cdots$ of moves in $W(x)$, such that, for each n, $R'(Y)_{\mathrm{Pre}(b_n)} < T'(x)$ (thus also $R(Y)_{\mathrm{Pre}(b_n)} \neq$ undef).

D tells us that the value of $R(Y)$, appearing in the array at $\mathsf{Ticket}(x)$, is read in x. W1 says that a permission to go is obtained by executing, for each Y, a successful read in $W(x)$, while W2 tells us that X may be prevented from

[2] by [Gur95] the official notation for these locations is $(\mathrm{mode}, X), (R, X), (A, (X,Y))$; since in the simple cases occurring in this paper, no ambiguity may arise, we shall use the applicative term notation as above also for locations.

going only by executing, for some $Y \neq X$, an infinite sequence of unsuccessful reads in $W(x)$, where a read $b \in W(x)$ from $R(Y)$ on behalf of X is successful if $R'(Y)_{\mathrm{Pre}(b)} > T'(x)$. It turns out that D, W1 and W2 is all that we need to know about reading actions in order to prove correctness and deadlock–freedom of $\mathcal{B}_1$.

Requirements D and W1 say that, for each Y, there is a move $b(Y)$ in x, respectively $W(x)$, having some property. Remark that, without loss of generality, we can assume that these moves $b(Y)$ are all distinct. Suppose namely that, in D or W1, we have $b = b(Y_1) = \cdots = b(Y_k)$. Then we can replace b with k distinct monitored moves which, in the partial order, follow and precede exactly the same moves as b does. It is easy to see that this replacement leaves us with a legitimate run, with exactly the same partial order of customers' moves. A remark of the same kind applies also to sequences of moves claimed by W3, but we shall not need that case.

The reader familiar with [BGR95] might notice that, what in similar requirements there were temporal conditions on some monitored locations, takes here (and in the next section) the shape of conditions on behaviour of unknown agents. The role of some time moments in proofs of [BGR95] thus turns out to be that of place holders for monitored moves.

5 Correctness and Deadlock–Freedom: The ASM $\mathcal{B}_2$

We define an ASM expressing a 'higher level' view of the Bakery Algorithm, similar to $\mathcal{B}_1$ but with the array abstracted away. The relevant datum to be described abstractly is the *ticket* assigned to a customer X (and written into its register $R(X)$) when X leaves the doorway and enters the wait section. We introduce for this purpose two monitored functions, boolean valued Ready and integer valued T, expressing, respectively, readiness of the ticket and its value.

The relevant moment to be analyzed is the moment at which a process with a ticket is allowed to enter the critical section. This 'permission to go' will also be represented by a monitored function, Go.

We will impose requirements on the environment and monitored moves, responsible for the values of Ready, T and Go, which will be shown to guarantee the correctness and deadlock–freedom of the higher level ASM $\mathcal{B}_2$. We will then show that these requirements are correctly implemented in $\mathcal{B}_1$.

5.1 The Program of $\mathcal{B}_2$

Start

```
if mode(me) = satisfied then
   R(me) := 1, mode(me) := doorway
```

Ticket

```
if mode(me) = doorway and Ready(me) then
   R(me) := T(me), mode(me) := wait
```

Entry

```
if mode(me) = wait and Go(me) then
   mode(me) := CS
```

Exit

```
if mode(me) = CS then
   mode(me) := done
```

Finale

```
if mode(me) = done then
   mode(me) := satisfied, R(me) := 0
```

5.2 Semantics of $\mathcal{B}_2$

The ASM $\mathcal{B}_2$ is similar to that of $\mathcal{B}_1$ except for the fact that the array is gone. In particular we assume the Progress Proviso (for known agents, i.e. customers). The role of the array is taken over by three monitored functions, Ready, T and Go. Looking at $\mathcal{B}_1$, Ready(X) and $T(X)$ can be seen as standing for the abbreviations used there, while Go(X) can be interpreted as the guard of the Entry rule, $\forall Y \neq X(A(X,Y) = 0$ or $(A(X,Y), id(Y)) > (A(X,X), id(X)))$.

The ASM $\mathcal{B}_2$ provides however no means to compute Ready, T and Go.

Our first requirement says that every interested customer eventually obtains his ticket:

C0 Each execution of Start, by a customer X, completes to a doorway x. For each x the value $T(X)_{\mathrm{Pre(Ticket(x))}}$ is indisputable.

The indisputable value of $T(X)_{\mathrm{Pre(Ticket(x))}}$ will be, like before, denoted by $T(x)$. In order to express the rest of our conditions on the array in terms of T and Go, we need some additional notation and terminology.

For open intervals in a partial order we also use $(a,b) < (c,d)$ if $b \leq c$, and say that the two intervals are concurrent if neither $b \leq c$ nor $d \leq a$. Note that concurrency does not necessarily imply overlap, i.e. existence of common elements; it in general just allows it.[3]

Sometimes we shall also compare elements with intervals: $c < (a,b)$ if $c \leq a$, likewise for $>$.

This ordering will help us to formalize the idea that tickets increase together with doorways (see C2 below). This should also apply in a way to concurrent

[3] Note however that, if intervals are interpreted as intervals on the partial order of initial segments, with (a,b) containing all segments containing a but not b, then concurrent intervals indeed overlap.

doorways; these are ordered by the following relation $\prec$, borrowed from its linear order analog of [Abr93].

Let $X \neq Y$, and let x, y range over doorways of X, Y respectively.

Definition 2. *$x \lhd y$ if x and y are concurrent and $T'(x) < T'(y)$. Further, $x \prec y$ if $x \lhd y$ or $x < y$.*

Lemma 4. *$x \prec y$ or $y \prec x$.*

Proof. Note that $T'(y) \neq T'(x)$ for $X \neq Y$, while two doorways of the same customer can never be concurrent. □

Our other conditions are then

C1 $T(x)$ is a positive integer > 1.
C2 If $y < x$ then either $\mathsf{Finale}(y) < \mathsf{Ticket}(x)$ or $T'(y) < T'(x)$.
C3 If $W(x)$ is complete, then, for every $Y \neq X$, there exists a move $b \in W(x)$, such that $T'(x) < R'(Y)_{\mathrm{Pre}(b)}$ (thus $R(Y)_{\mathrm{Pre}(b)} \neq$ undef).
C4 If $W(x)$ is incomplete, then there is a $y \prec x$ with $W(y)$ incomplete.

Intuitively, C2 says that tickets respect the temporal precedence of doorways with concurrent wait periods, C4 is an induction principle, and C3 expresses that permission to go is obtained by checking the ticket against competitors' registers. C2 (together with C0) is easily seen to be an abstract version of D, C3 is an abstract version of W1, while the fact, to be proved below, that C4 follows from W2 together with D,W1, is the essence of deadlock–freedom for the Bakery algorithm.

An immediate consequence of C3 is finite concurrency of doorways:

Corollary 2. *The set of doorways concurrent to any given doorway is finite.*

Proof. Let $x < x'$ be two doorways of X both concurrent to y. By C3 applied to x, there is a move b, with $x < b < x'$. Since $R(Y)_{\mathrm{Pre}(b)}$ is indisputable, by corollary 1 b compares to both ends of y; $b \leq \mathsf{Start}(y)$ would imply $x < y$, while $b \geq \mathsf{Ticket}(y)$ would imply $y < x'$, both contradicting the assumption of concurrency. Thus $b \in y$. But, by finite history, there can be only finitely many such b's. □

5.3 Correctness and fairness of $\mathcal{B}_2$

Lemma 5. (First Comes, First Served) *If $y \prec x$ and $W(x)$ is complete, then $W(y)$ is complete and $CS(y) < CS(x)$.*

Proof. Assume the premise is satisfied and the conclusion is false, i.e. that there is no move $\mathsf{Finale}(y) < \mathsf{Entry}(x)$. Take b as given by C3.

Claim 1 : $T'(y) < T'(x)$.

Claim 2 : $\mathsf{Ticket}(y) < b$.

Given the claims, we have $T'(y) < T'(x) < R'(Y)_{\mathrm{Pre}(b)} \neq$ undef, and thus Y must be writing to $R(Y)$ by a move in $(\mathsf{Ticket}(y), b)$. But the first such write after $\mathsf{Ticket}(y)$ must be a Finale move, which contradicts the assumption that the conclusion of the lemma is false.

Claim 1 follows immediately from definition of $\prec$ in case of concurrency, and from C2 otherwise.

To prove Claim 2, we first note that b is comparable to both ends of y, and, in view of $y \prec x$, $b \leq \mathsf{Start}(y)$ is impossible. It also impossible that $\mathsf{Start}(y) < b \leq \mathsf{Ticket}(y)$, since then $R(Y)_{\mathrm{Pre}(b)} = 1$, which contradicts the choice of b. □

Lemma 6. *$\prec$ is transitive.*

Proof. by contradiction. Suppose $x \prec y \prec z \prec x$. Count the number n of $<$'s in the above sequence of $\prec$ signs. In case $n = 0$ the statement follows from the fact that the order of integers (tickets) is transitive, and in cases $n = 2, 3$ the statement follows the fact that the partial order $<$ of open intervals in a partial order is transitive. In case $n = 1$, by symmetry, we may assume $x \lhd y \lhd z < x$ and therefore $T'(x) < T'(y) < T'(z)$. Given the assumption $x \prec y \prec z \prec x$, by Lemma 5, if one of the waiting sections is complete then so are the other two, and we have $\mathrm{CS}(x) < \mathrm{CS}(y) < \mathrm{CS}(z) < \mathrm{CS}(x)$ which is impossible. So all three waiting sections must be incomplete. Thus we can apply C2 to obtain also $T'(z) < T'(x)$, which is impossible. □

Lemma 7. (Deadlock freedom) *Every $W(x)$ is complete.*

Proof. By corollary 2 (and finite history) $\prec$ is well–founded. Then C4 is precisely the induction principle required to establish the claim. □

This section is summarized in the following

Theorem 1. *Doorways are linearly ordered by $\prec$. All waiting sections are complete, and $x \prec y$ implies $CS(x) < CS(y)$.*

5.4 $\mathcal{B}_1$ implements $\mathcal{B}_2$ correctly

We check that the requirements are satisfied in $\mathcal{B}_1$ (i.e. follow from D, W1, W2), where $\mathrm{Ready}(X) = (\forall Y \neq X(A(X,Y) \neq \text{undef}))$, $T(X) = 1 + \max_Y A(X,Y)$, and $\mathrm{Go}(X)$ means that the condition of the rule Entry is satisfied.

C0 is enforced by requirement D and the Progress Proviso for $\mathcal{B}_1$.

C1 is satisfied since the maximum in the rule Ticket is taken over all Y, including X which at that moment has register value $R(X) = 1$.

C2. Take b as given by D. Since $R(Y)_{\mathrm{Pre}(b)}$ is indisputable, the move b compares to all Start, Ticket and Finale moves of Y. With $\mathsf{Ticket}(y) \leq \mathsf{Start}(x) < b$, it is meaningful to ask whether Y executes the Finale move in $(\mathsf{Ticket(y)}, \mathsf{b})$. If it does, we are done; if it doesn't, $R(Y)_{\mathrm{Pre}(b)} = T(y)$, and, by D, $T'(y) < T'(x)$.

C3 follows immediately from W1.

C4. By contradiction, suppose that the premise is satisfied but the conclusion is false, i.e. $W(x)$ is incomplete but $W(y)$ is complete for all $y \prec x$. Let Y and $b_1 < b_2 < \cdots$ be the customer and the sequence of moves as given by W2.

Claim: There is a move $b \in W(x)$, with $R(Y)_{\mathrm{Pre}(b)} \neq$ undef, such that the following two properties hold for each y:

$$\text{(i) } b > \mathsf{Ticket}(y) \qquad \text{(ii) if } y \prec x \text{ then } b > \mathsf{Finale}(y).$$

First we derive the desired contradiction from the claim, and second we prove the claim.

So suppose that the claim is true and let b be as in the claim. Then $R(Y)_{\mathrm{Pre}(b)}$ has an indisputable value, and b thus compares to all moves of Y that change $R(Y)$. What is the value of $R(Y)$ in $\mathrm{Pre}(b)$? We have two possible scenarios. Scenario A: all $y \prec x$; then b succeeds every $\mathsf{Finale}(y)$ and thus $R(Y)_{\mathrm{Pre}(b)} = 0$. Scenario B: there is some y with $\mathsf{Ticket}(y) < b \leq \mathsf{Finale}(y)$; then $R(Y)_{\mathrm{Pre}(b)} = T(y)$. To summarize, if b is as in the claim, then $R(Y)_{\mathrm{Pre}(b)}$ is either 0 or $T(y)$, so that $R'(Y)_{\mathrm{Pre}(b)} \geq T'(y)$.

The values of $R(Y)_{\mathrm{Pre}(b)}$ and of $R(Y)_{\mathrm{Pre}(b_n)}$ for every n are indisputable. Moves b and b_n thus all compare with every move of Y which changes $R(Y)$. It is easy to see that any $b_n \not< b$ satisfies (i) and (ii) in the claim. But then, as shown above, $R'(Y)_{\mathrm{Pre}(b_n)} \geq T'(y)$, which contradicts the property of b_n in W2. Thus every $b_n < b$, which contradicts finite history.

It remains to prove the claim.

To prove the claim, note that, for $y \prec x$, $\mathrm{CS}(y)$ is defined and complete, by the assumption that $W(y)$ is complete and the Progress Proviso. It suffices to prove that there is at most one $y > x$. The sequence of doorways of Y is then finite: by finite history, it has finitely many elements $< x$, by corollary 2 finitely many elements concurrent to x. Thus Y has a last move, say e_Y. By Progress Proviso, e_Y can only be a Ticket or a Finale move. Since all b_n, by corollary 1, compare to e_Y, by finite history, for sufficiently large n we have $b_n > e_Y$. We can then, for claimed b, take any $b_n > e_Y$.

It remains to prove that Y has at most one doorway $> x$. Suppose $x < y$. Then, by C2 (with x, y playing y, x respectively), $T'(x) < T'(y)$ (since $W(x)$ is incomplete). If $W(y)$ were complete, by C3 there would be a $c \in W(y)$ such that $R'(X)_{\mathrm{Pre}(c)} > T'(y)$. But since $x < y < c$, we also have $c \in W(x)$ and $R(X)_{\mathrm{Pre}(c)} = T(x)$, so $T'(x) > T'(y)$, which is impossible. Thus $W(y)$ is incomplete, and y is the last doorway of Y.

We have thus verified that C0–C4 hold of arbitrary runs of $\mathcal{B}_1$. It follows that the results of the previous subsection, summarized in Theorem 1, hold of $\mathcal{B}_1$ as well.

6 Realizing the Model with Reading Agents: the ASM $\mathcal{B}_0$

6.1 The Program of $\mathcal{B}_0$

Until now the universe of known agents was populated by customers only. Now we also have another kind of agents. They present one way of making the unknown agents of $\mathcal{B}_1$ known.

Formally the universe of agents splits into two disjoint universes: Customer and Reader. Customers and readers are related by several functions. If X and Y are distinct customers, then in each state there is at most one reader–agent $r(X, Y)$, and there are no other readers.

If r is the reader $r(X, Y)$, then $\mathrm{Lord}(r) = X$ and $\mathrm{Subject}(r) = Y$. The readers will be created on the fly, when needed (at Start and Ticket moves), and will self–destruct when their task is completed.

Customer Start

```
if mode(me) = satisfied then
   A(me,me) := 1, R(me) := 1, mode(me) := doorway
   ∀ Y ≠ me create-reader(me, Y, doorway)
```

Ticket

```
if mode(me) = doorway and (∀Y ≠me) A(me,Y) ≠ undef then
   A(me,me) := 1 + max_Y A(me,Y), R(me) := 1 + max_Y A(me,Y)
   mode(me) := wait
   ∀ Y ≠ me create-reader(me, Y, wait)
```

Entry

```
if mode(me) = wait and
∀ Y ≠ me (A(me,Y)=0 or (A(me,Y),id(Y)) > (A(me,me),id(me))) then
   mode(me) := CS
```

Exit

```
if mode(me) = CS then
   mode(me) := done
```

Finale

```
if mode(me) = done then
   R(me) := 0, mode(me) := satisfied
   (∀ Y ≠ me) A(me,Y) := undef
```

where create-reader(X, Y, m) abbreviates the rule

```
create r
      agent(r) := true, Reader(r) := true
      program(r) := reader-program
      Lord(r) := X, Subject(r) := Y
      mode(r) := m
endcreate
```

Reader

```
A(Lord(me),Subject(me)) := R(Subject(me))
if mode(me) = doorway then
   destroy-reader(me)
if mode(me) = wait then
   if R(Subject(me)) = 0
      or (R(Subject(me)),id(Subject(me)))
         > (A(Lord(me),Lord(me)),id(Lord(me))) then
   destroy-reader(me)
```

where destroy-reader(a) abbreviates the rule

```
agent(a) := false, Reader(a) := false
program(a) := undef, Lord(a) := undef, Subject(a) := undef
```

6.2 Semantics of $\mathcal{B}_0$

Semantics of $\mathcal{B}_0$ is like that of $\mathcal{B}_1$, but considerably simpler, since all locations are controlled by the known agents, and there are no moves monitored by the known agents, to put constraints on—it is all in the programs for $\mathcal{B}_0$. The reader agents are one way to realize the requirement that those 'for-all commands in the doorway and the wait section of Lamport's pseudocode may be executed in many ways, in various sequences, all at once, concurrently etc.' In fact the reader agents capture all ways to realize that requirement, see below.

The only assumption we have to make, outside of the program, is the Progress Proviso, applying here to all agents, both customers and readers:

Progress Proviso. No reader, and no customer in mode other than *Satisfied*, stalls forever.

The reader–agents are created on the fly, and destroyed upon completion of their task: the effect of destroy-reader(a), if a is a reader–agent, is returning a to the reserve.

6.3 $\mathcal{B}_0$ realizes $\mathcal{B}_1$

The constraints D, W1, W2 can be read as a rather direct description of what the reader–agents do for their customers in $\mathcal{B}_0$. The fact that every run of $\mathcal{B}_0$ satisfies

D, W1, W2 follows from the programs and the Progress Proviso (together with the semantics described above or in [Gur95]).

D is satisfied in $\mathcal{B}_0$ since, for every move t of X executing Start, for every $Y \neq X$, there is a reader $r(X,Y)$ at Post(Start(x)). By programs and the Progress Proviso each of these readers makes a single self destructive move, which is the b required by D; by programs and the Progress Proviso X eventually executes Ticket.

By programs and Progress Proviso, for every $Y \neq X$ there is a reader $r(X,Y)$ at Post(Entry(x)). That reader makes a move in $W(x)$. For W1, W2 it then suffices to note

Fact 4. *A move t by $r(X,Y)$ in $W(x)$ is the last such move iff it is successful, i.e. $T'(x) < R'(Y)_{Pre(t)}$.*

W1 is namely satisfied in $\mathcal{B}_0$ since, for each $Y \neq X$, we can take the last waiting section move of $r(X,Y)$ for the claimed b.

W2 is satisfied in $\mathcal{B}_0$ since, if all $r(X,Y)$ for $Y \neq X$ have a last move in $W(x)$, by Progress Proviso X must eventually execute Entry. Thus for some $Y \neq X$ the reader $r(X,Y)$ keeps reading forever—take the sequence of his moves for $b_1 < b_2 < \cdots$ as claimed.

We have thus established that every run ρ_0 of $\mathcal{B}_0$ can be viewed as a run of $\mathcal{B}_1$. Since $\mathcal{B}_1$, as far as reading is concerned, can be viewed as a declarative description of algorithmic behaviour, rather then an algorithm proper, ρ_0 can also be seen as a realization of behaviour prescribed by $\mathcal{B}_1$.

To be more precise, let us introduce an appropriate implementation relation between moves and runs of the two ASMs.

A move t_0 by customer X in $\mathcal{B}_0$ *implements* a move t_1 by the same customer in $\mathcal{B}_1$ if the indisputable portions of states (values of $\mathrm{mode}(X), R(X), A(X,X)$) at $\mathrm{Pre}(t_0), \mathrm{Post}(t_0)$ coincide with those at $\mathrm{Pre}(t_1), \mathrm{Post}(t_1)$ respectively.

A run ρ_0 of $\mathcal{B}_0$ *implements* a run ρ_1 of $\mathcal{B}_1$ if the partial order of customers' moves in ρ_0 is order–isomorphic to the partial order of customers' moves in ρ_1, implementing it pointwise: whenever the isomorphism maps a move t_0 in ρ_0 to a move t_1 in ρ_1, then t_0 implements t_1.

In these terms, we have established that $\mathcal{B}_1$ (more specifically, D, W1, W2) provides a sound description of algorithmic behaviour: $\mathcal{B}_0$ is an algorithm behaving like that. For the record,

Lemma 8. (Soundness of $\mathcal{B}_1$) *Each run of $\mathcal{B}_0$ implements a run of $\mathcal{B}_1$.*

We can actually claim more. The requirements D, W1, W2 allow many different behaviours. Is there a behaviour, allowed by $\mathcal{B}_1$, which is not captured by the reader–agents of $\mathcal{B}_0$? Not really. This is the content of the following lemma, expressing a kind of completeness property.

Lemma 9. (Completeness of $\mathcal{B}_1$) *Each run ρ_1 of $\mathcal{B}_1$ is implemented by a run ρ_0 of $\mathcal{B}_0$.*

Proof. The idea is to transform ρ_1 to ρ_0 by implementing reading moves of ρ_1 with appropriate moves of reader–agents, possibly ignoring some inconsequential monitored moves of ρ_1. The replacement process is done in the order of ρ_1, that is earlier read moves are replaced (or discarded) earlier. The following conditions will be guaranteed by induction for every move b introduced by replacement, for every customer X and every doorway x of X. At $\mathrm{Pre}(b)$ there is a reader agent $r = r(X, Y)$ for some $Y \neq X$. If b is a move of r in x, then $\mathrm{mode}(r)_{\mathrm{Pre}(b)} =$ doorway (so that r self–destructs at b), and if b is a move of r in $W(x)$, then $\mathrm{mode}(r)_{\mathrm{Pre}(b)} =$ wait (so that r self–destructs at its last move in $W(x)$).

Now let $x = (a, b)$ be a doorway of X. By D, for each $Y \neq X$ there is a move $b(Y) \in x$ such that $A(X, Y)_{\mathrm{Pre}(\mathsf{Ticket}(x))} = R(Y)_{\mathrm{Pre}(b)}$. Recall that we can assume that $b(Y)$ are all distinct. Then implement each $b(Y)$ with a move of $r = r(X, Y)$. By induction condition, r is in mode doorway before $b(Y)$, and therefore self–destructs there.

The case of read moves in $W(x)$ is similar. Since $W(x)$ is complete, W1 guarantees that, for each $Y \neq X$ there is a move $b(Y) \in W(x)$ such that $R(Y)_{\mathrm{Pre}(b)} = A(X, Y)_{\mathrm{Pre}(\mathsf{Entry}(x))}$. Without loss of generality, all moves $b(Y)$ are distinct. Replace every $b(Y)$ with a move of $r = r(X, Y)$. By the induction condition, r is in mode wait before $b(Y)$, and therefore self–destructs at the move. If $b \in W(x)$ is a monitored move different from all $b(Y)$, we can discard it—also, if there is a $b(Y) > b$, we can implement b with an unsuccessful, nonselfdestructive read of $r(X, Y)$.

Finally, remove all remaining monitored moves in ρ_1. The result is the desired run ρ_0 of $\mathcal{B}_0$, implementing ρ_1. □

7 Concluding Remarks

Corrollary 2 implies the following *cofiniteness* property for the partial runs of the Bakery algorithm: for each move t the set $\{s | s \not> t\}$ is finite. This is a property of the Bakery Algorithm, and not of the modelling framework: in spite of finite history, it is easy to concoct legitimate partially ordered runs of some algorithm which violate the cofiniteness property. In the case of the Bakery Algorithm, the cofiniteness property implies that any two infinitely active customers have to synchronize infinitely many times.

The cofiniteness property is an example of a property of partial runs that is obfuscated in linear runs (since for linears runs it amounts to finite history). This indicates that concurrent computations may have significant properties which cannot be discovered by studying only their linearizations. Concurrent computations should be analyzed directly.

Acknowledgements

We thank Ante Djerek, Robert Eschbach and Igor Urbiha, who have provided very useful remarks on a draft version of this paper.

References

[Abr93] Uri Abraham. Bakery algorithms. Unpublished manuscript, pp. 35, 1993.

[BGR95] Egon Börger, Yuri Gurevich, and Dean Rosenzweig. The bakery algorithm: Yet another specification and verification. In E. Börger, editor, *Specification and Validation Methods*, pages 231–243. Oxford University Press, 1995.

[GM90] Yuri Gurevich and Larry Moss. Algebraic operational semantics and Occam. In E. Börger, H. Kleine Büning, and M. M. Richter, editors, *CSL'89, 3rd Workshop on Computer Science Logic*, number 440 in Lecture Notes in Computer Science, pages 176–192. Springer, 1990.

[GR93] Paola Glavan and Dean Rosenzweig. Communicating evolving algebras. In E. Börger, H. Kleine Büning, G. Jäger, S. Martini, and M. M. Richter, editors, *Computer Science Logic*, number 702 in Lecture Notes in Computer Science, pages 182–215. Springer, 1993.

[Gur95] Yuri Gurevich. Evolving algebra 1993: Lipari guide. In E. Börger, editor, *Specification and Validation Methods*, pages 9–36. Oxford University Press, 1995.

[Lam74] Leslie Lamport. A new solution of Dijkstra concurrent programming problem. *Communications of the ACM*, 17(8):453–455, 1974.