Relationships between communication models in networks using atomic registers

Lisa Higham\(^1\) and Colette Johnen\(^2\)

\(^1\)University of Calgary  
Computer Science Department  
Calgary, Alberta, Canada  
higham@cpsc.ucalgary.ca

\(^2\)Université de Paris-Sud  
LRI - CNRS  
Orsay, France  
colette@lri.fr

Abstract

A distributed system is commonly modelled by a graph where nodes represent processors and there is an edge between two processors if and only if they can communicate directly. In shared-registers versions of this general description, neighbouring processors communicate by reading or writing shared registers, where each read or write is one atomic step. Variants of shared register models occur in the literature. This paper defined two models of shared registers determined by selecting the register locations. In the atomic-state model each processor has a register; in the atomic-link model, each communication link has a register.

We determine under what conditions and with what robustness and/or failure-tolerance guarantees it is possible to transform a solution under the atomic-state model into a solution under the atomic-link model. The fault-tolerance models considered in this paper are wait-freedom and self-stabilization.

These questions are addressed by first establishing a framework for defining correct transformations, which may be useful for similar studies of the relationship between various models of distributed computation.

Keywords: distributed algorithms, communication models, shared atomic registers, single-readers, multi-readers, wait-freedom, self-stabilization.

1. Introduction

Network communication models  There is a proliferation of network communication models for distributed computing consisting of both shared memory and message passing paradigms. Different communities adopt different variants as the “standard” model for their research setting. Some are less realistic but support easier reasoning; others more closely capture reality but are harder to work with. In the first paper on self-stabilizing distributed algorithms \([5]\), Dijkstra assumed that in a network, each processor could read the state of each of its neighbours and update its own state in one atomic step. Let us call the model used by Dijkstra the composite state model. Dolev, Israeli and Moran \([7]\) introduced a read/write atomicity model for self-stabilizing algorithms to better capture the actual possible communication between processors. In their model, for each link between two processors, there are two single-writer/single-reader atomic registers, each one writable by one processor and readable by the other \([7, 6]\). This model can be used to simulate a message passing setting. Let us call this model the atomic-link model.

Many subsequent papers have used similar atomic register models. Furthermore, much research has been dedicated to constructing compilers that translate programs designed for the composite state model to programs that are correct and efficient assuming only read/write atomicity. There is, however, an important distinction between the two variants of read/write atomicity assumed in the self-stabilization literature. In several papers, the read/write atomicity model assumes that a single-writer/multi-reader atomic register resides at each processor and each processor owns the registers that it holds \([17]\). Each such register is writable by the owner and readable by each neighbour of the owner. Let us call this model the atomic-state model. In both the atomic-state and atomic-link models, an atomic step by a processor consists of either reading or writing one of the available registers. We are interested in the differences between the atomic-state and atomic-link models, and in determining the
existence of compilers between these two models.

As the size and complexity of networks increases, the likelihood of failure of a component somewhere in the system increases. This motivates us to design algorithms (and compilers) that have built-in fault-tolerance. The fault-tolerant models considered in this paper are wait-freedom and self-stabilization.

Informally, an operation is wait-free if no processor invoking the operation can be forced to wait indefinitely for another processor [9]. Such robustness implies that stopping failures (or very slow execution) of any subset of processors cannot prevent another processor from correctly completing its operation. Since unbounded waiting is prohibited, wait-free algorithms are necessarily lock-free. Wait-free implementations avoid well-known problems such as deadlock and livelock. Most of the research on wait-free implementations, however, assumes a globally shared memory model, where each processor can read and write any register, which is substantially stronger than either the atomic-state or atomic-link network models.

Informally, a self-stabilizing system is guaranteed to converge to the intended behavior in finite time, regardless of the initial states of either the processors or the communication registers. An algorithm is self-stabilizing if, after a burst of transient errors of some components of a distributed system (which leaves the system in an arbitrary configuration), the system recovers and returns to the specified configurations. If a self-stabilizing algorithm is general enough, it can also deal with topology change, so that the system will still automatically converge to eventually have a correct behavior as the network topology changes over time.

Related research There are several papers [2, 17] that provide self-stabilizing compilers from the composite state model to atomic-state model for various sets of network topologies. Antonioiu and Srimani's compiler [2] applies to general topologies where processors have distinct identifiers. It depend on unbounded timestamps. The same paper also presents a self-stabilizing compiler for any spanning tree network that uses bounded timestamps. To ensure safety, Antonioiu and Srimani's compilers require that no processor enters the critical section while the timestamps are "wrapping around". Nesterenko and Arora's compiler [17] is based on a bounded space self-stabilizing dining philosophers protocol for systems with atomic-state registers. The processors require distinct identifiers and every processor has to participate even if its does not want to perform an operation.


Wait free (but not self-stabilizing) transformation from one register model to another one have been extensively studied [1, 8, 16, 3]. Hoepman, Papatriantafiol and Tsingas [12] presented self-stabilizing versions of well-known implementations of shared register. For instance, they present a wait-free self-stabilizing implementation of a multi-writer/multi-reader atomic register using single-writer/dual-reader regular registers of unbounded size. These implementations require globally shared memory.

In the full version of this paper[11], we study the relationships between four different models: the atomic-state, and the atomic-link models and the two corresponding models where the registers are only regular rather than atomic. We determine the existence or not of wait-free compilers between these models. We present a self-stabilizing compiler from any of these four models to any other one.

Paper organization Section 2 defines the communication models atomic-state and atomic-link, and the fault-tolerance requirements wait-freedom and self-stabilization. We formalize the notion of a compiler from a network using one communication model to the same network topology where communication assumes a different model. The brief Section 2.2 lists some relationships between wait-freedom and self-stabilization that are exposed by their formal definitions.

Sections 3 and 4 present our main impossibility and possibility results respectively. Section 3 establishes that there is no general wait-free compiler from atomic-state networks to atomic-link networks. The proof proceeds by showing that any such compiler would require shared registers between any two processors, which is not the case in general networks. The proof relies heavily on the proof ([3], page 222) that in any wait-free construction of a single-writer/multi-reader atomic register from single-writer/single-reader atomic registers, some reader must write. In section 4, we present a self-stabilizing compiler from networks where neighbours communicate via atomic-state registers to systems where neighbours communicate via atomic-link registers.
2. Definitions

2.1. Distributed systems

Shared registers Let $R$ be a single-writer/multi-reader register that can contain any value in domain $T$. $R$ supports only the operations READ and WRITE, each of which has a time interval corresponding to the time between the operation invocation and its response. Because there is only one writer, WRITE operations to $R$ happen sequentially, so they cannot overlap. READ operations, however, may overlap each other and may overlap a WRITE. Lamport [15] defined three types of such registers depending on the semantics when READ and WRITE operations overlap. Let $I$ be a set of READ and WRITE operations labelled with their time intervals. Register $R$ is atomic if (i) each READ that does not overlap any WRITE returns the value of the most recent preceding WRITE, and, (ii) if a READ overlaps a WRITE and returns the value being written, then any subsequent READ that overlaps the same WRITE must return the value of a preceding WRITE. A sequence of READ and WRITE operation intervals on an atomic register is valid for atomic registers (or just valid) if each READ interval in the sequence satisfies this condition. This definition of validity is equivalent to the definition of Linearizability [10] of read and write operations on registers.

Network models A distributed network can be modelled by a graph $G = (V, E)$ where $V$ is a set of processors and an edge $(pq) \in E$ if and only if processors $p$ and $q$ can communicate directly. Several variants have been defined depending on the precise meaning of “communicate directly”. In this paper we consider two variants where each processor uses a collection of local registers accessible only to itself and communicates with its neighbours via shared registers. The way these registers are shared distinguishes the models.

In the atomic-state network model, each processor $p$ owns a single-writer multi-reader shared atomic register $R_p$, which is writable by $p$ and readable by each of the $p$’s neighbours. In one step a processor can either read an atomic register of one of its neighbours (storing its value into its own local variables) or write its own shared atomic register. In an atomic-state network model, the WRITE and READ operations are denoted:

- $\text{ATOMIC-STATE-WRITE}(R, v)$ to denote the write of value $v$ to the shared register $R$.
- $v \leftarrow \text{ATOMIC-STATE-READ}(R)$ to denote the read of the shared register $R$ that returns the value $v$.

In the atomic-link network model, for each edge $(pq) \in E$, there are two single-writer, single-reader atomic registers. Register $R_{pq}$ is writable by $p$ and readable by $q$; register $R_{qp}$ is writable by $q$ and readable by $p$. In one atomic step a processor can either read one of the shared registers to which it has read access, or write a shared register to which it has write access. The atomic-link model is identical to a model used by Dolev,Israeli and Moran [7]. In an atomic-link network model, the WRITE and READ operations are denoted $\text{ATOMIC-LINK-WRITE}(R, v)$ and $v \leftarrow \text{ATOMIC-LINK-READ}(R)$ respectively.

Distributed algorithms, distributed systems A distributed algorithm is an assignment of a program to each processor in the network; this assignment gives rise to a distributed system. The assigned program must use only the register types and operations available in the network model.

Configurations and computations A configuration of a distributed system is a collection of values assigned to all the registers of the system. Given some non-empty subset of processors, $S$, and a configuration $C$, the configuration $S(C)$ arises when, starting in $C$, all processors in $S$ simultaneously execute the next step of their programs. A schedule is a sequence of non-empty subsets of processors. The computation that arises from a schedule $S = S_1, S_2, \ldots$ and a starting configuration $C_0$ is the sequence of configurations $C = C_0, C_1, \ldots$ where $C_i = S_i(C_{i-1})$ for $i \geq 1$. A Scheduler is any collection of schedules.

Some Schedulers are of particular interest. The Unfair Scheduler has no requirement to eventually select each processor. A Scheduler is fair if, in every infinite computation, every processor executes an infinite number of steps.

Distributed problems and solutions Without loss of generality we assume that a distributed computation problem is specified as a predicate over computations. A (deterministic) distributed algorithm $A$ solves problem $P$ for network class $N$ if for any network $N \in N$ all computations of algorithm $A$ on $N$ satisfies predicate $P$.

2.2. Fault-tolerance

A operation on a shared object is wait-free if every invocation of the operation completes in a finite number of steps of the invoking processor regardless of the number of steps taken by any other processor.
Let \( \mathcal{L} \) be a predicate defined on configurations. A distributed system is **self-stabilizing to \( \mathcal{L} \)** if and only if

- **convergence**: starting from any configuration, any computation reaches a configuration satisfying \( \mathcal{L} \).
- **closure**: from any configuration \( C \) satisfying \( \mathcal{L} \) the next step under any computation satisfies \( \mathcal{L} \).

The predicate \( \mathcal{L} \) is called a **legitimacy predicate** and when the system has converged to a configuration satisfying \( \mathcal{L} \) we say it has **stabilized**.

A self-stabilizing system cannot terminate, because otherwise it is possible that at termination a fault occurs, which would never be detected and thus not corrected.

**Some Relationships Between wait-freedom, and self-stabilization**

Some relationships are exposed by examining the safety and liveness requirements of the fault-tolerance models considered here (wait-freedom and self-stabilization).

A **self-stabilizing system** requires:

- **safety**: Safety (closure to configurations satisfying the legitimacy predicate) is required eventually regardless of the configuration in which the algorithm begins.
- **liveness**: System liveness (convergence to the legitimacy predicate) is required under a set of schedules.

A **wait-free implementation** of an object requires:

- **safety**: Safety is required always provided the algorithm begins in one of the specified initial configurations.
- **liveness**: Unconditional liveness is required always. Individual progress is required regardless of the participation of other processors.

A **wait-free self-stabilizing system** requires:

- **safety**: Safety (closure to configurations satisfying the legitimacy predicate) is required eventually regardless of the configuration in which the algorithm begins.
- **liveness**: Unconditional liveness is required always. Individual progress is required regardless of the participation of other processors.

Observe that a wait-free self-stabilizing system requires the safety of self-stabilization and the liveness of wait-freedom.

**Schedulers, wait-freedom, and self-stabilization**

Schedulers can be used to describe wait-freedom. The unfair scheduler is unrestricted as to what set of processors it chooses at each step. Thus, in these models, any algorithm that is self-stabilizing under the unfair scheduler, is also wait-free.

**Observation 2.1** For any atomic-state or atomic-link system, self-stabilization under the unfair Scheduler implies wait-free self-stabilization under the unfair Scheduler.

### 2.3. System transformations and compilers

A transformation of a system on a specified network model to a system on another network model (called the target model) is achieved by transforming each operation available at the specification level to a program of operations available in the target model. This paper is concerned with program transformations from atomic-state networks to atomic-link networks. Let \( G \) be a graph; denote by \( \text{AS}(G) \) the atomic-state network with topology \( G \), and denote by \( \text{AL}(G) \) the atomic-link network with topology \( G \). To transform an algorithm for \( \text{AS}(G) \) to an algorithm for \( \text{AL}(G) \) we replace each **atomic-state-write** and **atomic-state-read** by every processor \( p \) in \( \text{AS}(G) \) with a program for \( p \) in \( \text{AL}(G) \) that uses only local operations and the operations **atomic-link-write** and **atomic-link-read**. Thus a **program transformation** from \( \text{AS}(G) \) to \( \text{AL}(G) \) is just a mapping, \( \tau \), where \( \tau(\text{atomic-state-write}(R, v)) \), and \( \tau(\text{atomic-state-read}(R)) \) are programs whose operations are on registers in \( \text{AL}(G) \) and such that \( \tau(\text{atomic-state-read}(R)) \) returns a value. We desire these program transformations to preserve correctness. Since correctness is defined by a predicate on computations or/and on configurations, and the computations and configurations differ in each network model, we need to make precise what is meant for \( \tau \) to "preserves correctness".

Let \( \mathcal{A} \) be an program for \( \text{AS}(G) \). A computation \( C \) of \( \tau(\mathcal{A}) \) on \( \text{AL}(G) \) is **Linearizable** if the collection of operation in \( C \) are valid for atomic registers. It is straightforward to check that this correctness condition agrees with **Linearizability** as used by Lamport [13] and named and used by Herlihy and Wing [10]. That is, for a Linearizable computation, there is a **linearization point** for each **write** and **read** operation \( o \) by \( \mathcal{A} \) between the invocation and response of \( \tau(o) \) such that, with operations ordered according to their linearization point, each **read** returns the value of the most recent preceding **write** to the same register. The algorithm \( \tau(\mathcal{A}) \) **implements** \( \mathcal{A} \) on \( \text{AL}(G) \) if every computation of \( \tau(\mathcal{A}) \) is Linearizable; in this case \( \tau(\mathcal{A}) \) is an **implementation of** \( \mathcal{A} \) on \( \text{AL}(G) \).
A compiler from $AS(G)$ to $AL(G)$ is a transformation that implements every algorithm for $AS(G)$ on $AL(G)$. A transformation is a self-stabilizing compiler if it is a compiler and it maps self-stabilizing systems to self-stabilizing systems. A compiler is wait-free if it maps wait-free algorithms to wait-free algorithms.

3. Impossibility of a Wait-free Compiler from Atomic State Systems to Atomic Link Systems

Let $G$ be any connected graph. Given an algorithm $Alg$ for an atomic-state network $AS(G)$, we would like to implement it on the atomic-link network $AL(G)$. Attiya and Welch ([3] page 336) provide a wait-free compiler for this task provided the network $G$ is a complete graph. Also there are existing implementations of a multi-reader register by single-reader registers [4, 13, 18, 19] and it is straightforward to convert these to a compiler from atomic-state to atomic-link provided the network is complete. Furthermore, the most sophisticated of these implementations [3] use bounded time-stamps to ensure that these implementations use only bounded size single-reader registers provided the original multi-reader registers have bounded size. In this section, we show that if $G$ is not a complete graph, then there is no compiler that can do this conversion in a wait-free manner.

The relationship between the atomic-state and atomic-link models is similar to the relationship between single-writer/multi-reader registers and single-writer/single-reader registers. We first show any shared memory wait-free implementation of a single-writer/multi-reader register from a collection of single-writer/single-reader registers must have a register shared between each pair of readers.

Attiya and Welch ([3] page 222) show that in any wait-free construction of a single-writer/multi-reader atomic register from single-writer/single-reader atomic registers, some reader must write. In fact, all constructions in the literature employ a shared register between each pair of readers. The next claim shows that, as conjectured by Lampert [15], communication between each pair of readers is necessary. The proof is by contradiction; it constructs a computation that cannot be linearized. The technique is inspired by that of Attiya and Welch. There are now writes occurring by the readers as well as the writers, however, which can influence the writer’s behavior. Thus one cannot fix in advance the sequence of writes by the writer. Instead we construct the required computation as the execution proceeds.

**Lemma 3.1** Any wait-free implementation of a single-writer/multi-reader atomic register from single-writer/single-reader atomic registers must have a single-writer/single-reader register shared between each pair of readers.

**Proof:** Let $\mathcal{R}$ be the single-writer/multi-reader atomic register to be implemented, and let $w$ denote the writer. Denote the write and read operations to $\mathcal{R}$ by write and read respectively. Denote by write and read, the operations on the single-writer/single-reader registers of the implementations. By way of contradiction, suppose $p$ and $q$ are any two readers that do not share any register. Suppose the initial value of $\mathcal{R}$ is 0. We construct a computation that has $p$ and $q$ repeatedly executing read of $\mathcal{R}$ while $w$ executes a single write of value 1 to $\mathcal{R}$. No processes other than $w$, $p$ and $q$ access $\mathcal{R}$ during this interval. The computation will have some read return the old value 0, after an earlier read returns the new value 1, providing the required contradiction.

First form a partial execution, $E$, inductively as follows. Initially $E$ is empty and has 0 segments. Extend $E$ a segment at a time, by, at each step, letting $w$ run alone until it has executed exactly one (more) write in its program for write. Then pause $w$ and sequentially execute a complete read of $\mathcal{R}$ by $p$, followed by a non-overlapping and complete read of $\mathcal{R}$ by $q$. Because read of $\mathcal{R}$ is a wait-free operation, it can be performed in between two write operations. The partial execution $E$ consists of all segments up to but not including the first segment where either $p$ or $q$ returns the new value, 1. Since the write by $w$ is wait-free, it will eventually complete. After that, all subsequent read operations must return 1 to be correct. So eventually $p$ or $q$ must return 1. Thus $E$ has a finite number of segments, and in every segment of $E$ both $p$ and $q$ return 0 for their read operations.

Now construct two alternative extensions of $E$ by one more segment. In the first, $E$ is extended to $E1$ by letting $w$ run alone until it has executed exactly one (more) write. Then pause $w$ and sequentially execute a complete read of $\mathcal{R}$ by $p$, followed by a non-overlapping and complete read of $\mathcal{R}$ by $q$, followed by letting $w$ finish its write to completion while executing alone. From the construction of $E$, in computation $E1$ either $p$ or $q$ returns 1 for its read in this last segment.

In the second, $E$ is extended to $E2$ in nearly the same way except that the ordering of $p$ and $q$ reversed. That is, add one more segment by letting $w$ run alone until it has executed exactly one more write in its program for write, followed by a read of $\mathcal{R}$ by $q$, and...
then a non-overlapping read of \( R \) by \( p \), followed by letting \( w \) finish its write to completion while executing alone.

Since the write by \( w \) at the beginning of the last segment is to a single-writer/single-reader register, it can be read by at most one of \( p \) and \( q \), and cannot be overwritten by either. Since \( p \) and \( q \) do not share any registers, and no other processors are participating, \( p \) and \( q \) have no information other than this one write by \( w \) that is different in the last segment from the preceding segment. So for at least one of \( p \) and \( q \), there is no write that has occurred since it executed its read in the second last segment that is visible to it. For this processor, the last two segments are indistinguishable. Hence, this process will again return 0 for its read.

For each processor \( p \) and \( q \), and for any segment \( i \), its state and the values of all its shared variables at the beginning of its computation in segment \( i \) are identical in both \( E1 \) and \( E2 \), so \( E1 \) and \( E2 \) are indistinguishable to either of \( p \) or \( q \). Thus, in every segment of \( E2 \), each processor will return the same value as it did in the corresponding segment of \( E1 \). Hence, one returns 1 and the other returns 0 in the last segment. If \( p \) returns 1 and \( q \) returns 0, then computation \( E1 \) fails to implement the atomic register \( R \) because it contains two non-overlapping reads where an old value of the register is returned after a new value. If \( q \) returns 1 and \( p \) returns 0, then computation \( E2 \) fails for the same reason.

**Theorem 3.2** If \( G \) is any network topology that is not complete, then there is no wait-free compiler from \( AS(G) \) to \( AL(G) \).

**Proof:** Let \( p \) and \( q \) be two processors that are separated by distance 2 in \( G \) and let \( w \) be a neighbour of both \( p \) and \( q \).

Consider the operations \( \text{atomic-state-write}(R_w, v) \) and \( \text{atomic-state-read}(R_w) \) of a single-writer/multi-reader register \( R_w \) owned by \( w \) and shared with its neighbours in \( AS(G) \). If there is a wait-free compiler that transforms an algorithm on \( AS(G) \) to an algorithm \( AL(G) \), then it must compile these \( \text{atomic-state-write} \) and \( \text{atomic-state-read} \) operations into programs that use the \( \text{atomic-link-read} \) and \( \text{atomic-link-write} \) operations available to \( w \), \( p \) and \( q \) in \( AL(G) \). Since each of these link-registers is a single-writer/single-reader register, this compiler implements the multi-reader register \( R_w \) using single-reader registers. By Lemma 3.1, any such implementation requires a shared register between \( p \) and \( q \), which does not exist in \( AL(G) \). Thus there is no wait-free compiler from the from \( AS(G) \) to \( AL(G) \).

**4. A Self-stabilizing Compiler from Atomic State Systems to Atomic Link Systems**

Let \( \mathcal{A} \) be the set of algorithms for the atomic-state model that satisfy:

(i) every processor reads each of its in-registers infinitely often, and

(ii) every processor writes its out-registers at least two times during the stabilization time.

We show that Algorithm 1 is a self-stabilizing compiler from atomic-state networks to atomic-link networks for all algorithms in \( \mathcal{A} \).

The self-stabilizing communication primitives \texttt{acknowledged\_writing} and \texttt{acknowledged\_reading} for the atomic-link model appeared earlier [14]. These primitives ensure that a processor writes a new value in its registers only after that all its neighbours have read the previously written values. This reliable transfer of communication variables from neighbouring processors \( p \) to \( q \) is achieved as follows. The register \( R_{qp} \) has \( 2 \cdot k \) fields where \( k \) is the number of communication variables. Two fields called \texttt{local}_x and \texttt{copy}_x are associated with each communication variable \( x \). The \texttt{local}_x field contains the value of variable \( x \) that \( q \) wants to communicate to \( p \). The \texttt{copy}_x field contains the last read value of \( p \)'s variable \( x \) by \( q \). During a reading operation by \( p \) of register \( R_{qp} \), \( p \) copies the values of all \texttt{local} fields of \( R_{qp} \) into the \texttt{copy} fields of the register \( R_{pq} \). After a writing operation, \( p \) checks to determine if the value of each \texttt{copy} field of register \( R_{qp} \) is equal to the local value of the associated communication variable. If this checking succeeds, \( q \) has the latest values from \( p \) of all the communication variables, so the local variable \texttt{ok\_q} is set to 1. Once all \( p \)'s neighbours have read the new values of communication variables the \texttt{acknowledged\_writing} by \( p \) is over. Observe that \texttt{acknowledged\_reading} is not blocking. The following Claim is proved in earlier work [14].

**Claim 4.1** ([14]) Assuming that each processor performs \texttt{acknowledged\_reading} infinitely often, any execution of \texttt{acknowledged\_writing} eventually completes.

The communication variables for Algorithm 1 are, for each processor, the state variables (called \texttt{state}) used in the algorithm \( \mathcal{A} \), plus a flag value (called \texttt{flag}).

During the second complete execution of the \texttt{acknowledged\_writing} by \( p \) with distinct flags, all its neighbours perform an \texttt{atomic-link-write} operation.
Algorithm 1 Self-stabilizing compiler from atomic-state systems to atomic-link systems

structure of a register:
\( R = (local\_state, local\_flag, copy\_state, copy\_flag) \)
where local\_flag and copy\_flag fields have boolean values; local\_state and copy\_state fields have state values of the specified algorithm.

local Variables on \( p \):
flag - boolean variable
state - state variable of the specified algorithm
\( \forall r \in N.p, (N.p is the neighbours set of p) \),
ok\_r - boolean variable
\( L_{Reg}_p \) and \( L_{Reg}_p \) - same structure as \( R \)

code on the processor \( p \):
\[ \tau(\text{atomic-state-write})(R_p, \text{new state}) \]
\[ \text{state} := \text{new state}; \]
\[ \text{flag} := 0; \text{acknowledged\_writing(state)} \] [I]
\[ \text{flag} := 1; \text{acknowledged\_writing(state)} \] [II]

\[ \tau(\text{atomic-state-read})(R_q) \]
\[ \text{repeat} \]
\[ \text{for } r \in N.p \text{ do} \]
\[ \text{acknowledged\_reading}(R_{rp}); \]
\[ \text{done} \]
\[ \text{until } L_{Reg}_p, \text{local\_flag} = 1 \]
\[ \text{return } L_{Reg}_p, \text{local\_state} \]

\[ \text{acknowledged\_writing(state);} \]
\[ \text{for } r \in N.p \text{ do} \]
\[ \text{acknowledged\_reading}(R_{rp}); \text{ok\_r} := 0; \]
\[ \text{done} \]
\[ \text{repeat} \]
\[ \text{for } r \in N.p \text{ do} \]
\[ \text{acknowledged\_reading}(R_{rp}); \]
\[ \text{if } (L_{Reg}_p, \text{copy\_state} = \text{state}) \]
\[ \wedge (L_{Reg}_p, \text{copy\_flag} = \text{flag}) \text{ then} \]
\[ \text{ok\_r} := 1; \]
\[ \text{done} \]
\[ \text{until } (\forall r \in N.p, \text{ok\_r} = 1) \]

\[ \text{acknowledged\_reading}(R_{rp}); \]
\[ L_{Reg}_p \leftarrow \text{atomic-link-read}(R_{rp}) \]
\[ L_{Reg}_p, \text{local\_state} = \text{state}; \]
\[ L_{Reg}_p, \text{local\_flag} := \text{flag}; \]
\[ L_{Reg}_p, \text{copy\_state} := L_{Reg}_p, \text{local\_state}; \]
\[ L_{Reg}_p, \text{copy\_flag} := L_{Reg}_p, \text{local\_flag}; \]
\[ \text{atomic-link-write}(R_{pr}, L_{Reg}_p); \]

This operation may not be inside a complete execution of the acknowledged\_reading primitive. During the third complete execution of the acknowledged\_writing by \( p \) with distinct flags, all \( p \)'s neighbours perform a atomic-link\_write operation inside a complete execution of the acknowledged\_reading primitive. Thus, at the end of three complete executions of the acknowledged\_writing primitive by \( p \) with distinct flags, for any neighbour \( q \) of \( p \), \( L_{Reg}_p, \text{copy\_state} = L_{Reg}_p, \text{local\_state}(q) = \text{state}(p) \) and \( L_{Reg}_p, \text{copy\_flag} = L_{Reg}_p, \text{local\_flag}(q) = \text{flag}(p) \) (see [14] for a formal proof).

Lemma 4.2 For any algorithm \( \text{Alg} \) in \( A \), any execution of \( \tau(\text{atomic-state-write}) \) by any processor \( p \) eventually terminates.

Proof: The lemma follows immediately from the code for \( \tau(\text{atomic-state-write}) \) and Claim 4.1 and the properties of \( A \).

Definition 1 Consider the \( i \)th execution of \( \tau(\text{atomic-state-write}) \) by processor \( p \).
Let st\((i,p)\) denote the start time.
Let et\((i,p)\) denote the end time.
Let \( mt(i,p) \) denote the time that line [I] has completed and line [II] has not begun
The value of state\((p)\) during the \( i \)th execution of \( \tau(\text{atomic-state-write}) \) by \( p \) is denoted st\(_i\)p.

Observation 4.3 At time \( mt(1,p) \) for any neighbour \( q \) of \( p \), we have: \( R_{pq}, \text{local\_state} = \text{state}(p) = \text{st}_{1.p} \) and \( R_{pq}, \text{local\_flag} = \text{flag}(p) = 0 \).
At time \( et(1,p) \) for any neighbour \( q \) of \( p \), we have: \( L_{Reg}_p, \text{copy\_state}(q) = \text{state}(p) = \text{st}_{1.p} \) and \( L_{Reg}_p, \text{copy\_flag}(q) = \text{flag}(p) = 1 \).
At time \( mt(i,p) \) and \( et(i,p) \), for \( i \geq 2 \), for any neighbour \( q \) of \( p \), we have: \( L_{Reg}_p, \text{copy\_state}(q) = L_{Reg}_p, \text{local\_state}(q) = \text{state}(p) = \text{st}_{i.p} \) and \( L_{Reg}_p, \text{copy\_flag}(q) = L_{Reg}_p, \text{local\_flag}(q) = \text{flag}(p) \).

Lemma 4.4 Any execution of \( \tau(\text{atomic-state-read}) \) eventually terminates.

Proof: Let \( p \) and \( q \) be two neighbouring processors. If \( q \) executes \( \tau(\text{atomic-state-write}) \) a finite number of times, then \( L_{Reg}_p, \text{copy\_flag}(p), L_{Reg}_p, \text{local\_flag}(p) \), and \( \text{flag}(q) \) will eventually keep the value 1 forever. After that time, the execution of \( \tau(\text{atomic-state-read}) \) by \( p \) consists only of \( |N.p| \) atomic read operations. Therefore, any execution of \( \tau(\text{atomic-state-read}) \) eventually terminates.

Assume that \( q \) executes \( \tau(\text{atomic-state-write}) \) infinitely often. Let \( t' \) be the starting time of an
execution of $\tau$(atomic-state-read) by $p$. Let us call $t$ the next starting time of the execution by $q$ of $\tau$(atomic-state-write) after $t'$. Without lost of generality, we can assume that was the ith call of $\tau$(atomic-state-write) by $q$.

Assume that $i > 1$. Processor $p$ executes acknowledged_reading($Reg_{qp}$) at least once during the time interval $[mt(i,q), et(i,q)]$. At the end of this execution $L_{Reg_{qp}} copy \_flag(q) = 1$. $p$ executes the primitive acknowledged_reading($Reg_{qp}$) at least once during the time interval $[et(i,q), mt(i+1,q)]$. At the end of this execution $L_{Reg_{qp}} copy \_flag(q) = 0$. Within the time interval $[mt(i,p), mt(i+1,q)]$, $p$ performed the test $T$ at least once at the time when $L_{Reg_{qp}} local \_flag(p) = 1$. - between the two executions of the primitive acknowledged_reading($Reg_{qp}$) -. Thus, the execution of $\tau$atomic-state-read of $p$ terminates before the time $mt(i+1,q)$ or before the time $mt(3,q)$ (if $i = 1$).

Linearization points: The linearization point of the ith call of $\tau$atomic-state-write by $p$ is the time $mt(i,p)$ (where $i > 1$). The linearization point of a $\tau$atomic-state-read is its ending time. According to Theorem 4.7, each $\tau$atomic-state-read of the p’s state that terminates after the time $et(1,p)$, returns the written state of the preceding call of $\tau$atomic-state-write by $p$.

Lemma 4.5 Let $q$ and $p$ be two neighbouring processors. Let $i > 1$. Let $t$ be a time where a call of $\tau$atomic-state-read of p’s state by $q$ terminates. If $mt(i,p) < t < mt(i+1,p)$ then $\tau$atomic-state-read returns the value st.i.p.

Proof: For $i > 1$, during the time interval $[mt(i,p), mt(i+1,p)]$, any neighbour $q$, of $p$ verifies the following predicate: $(L_{Reg_{qp}} \_local \_state(q) = st.i.p \lor L_{Reg_{qp}} \_local \_flag(q) = 0)$. Thus $q$ can only get the value st.i.p during time interval $[mt(i,p), mt(i+1,p)]$.

Definition 2 Let $p$ and $q$ two neighbouring processors. We denote by wrong-read a call of $\tau$atomic-state-read to get p’s state that (i) does not return st.1.p and (ii) that terminates during the time interval $[mt(1,p), mt(2,p)]$.

Lemma 4.6 A wrong-read of p’s state terminates before time $et(1,p)$.

Proof: At the end of the execution of the primitive acknowledged_reading to read $R_{pq}$ which terminates after the time $et(1,p)$, we have $L_{Reg_{qp}} copy \_state(q) = st.i.p$ where $i \geq 1$ - see Observation 4.3.

Let $r$ be a wrong-read of p’s state by $q$. Let $t_r$ be the ending time of the last call to acknowledged_reading during this read operation, $r$, of $R_{pq}$. At time $t_r$, we have $L_{Reg_{qp}} copy \_state(q) \neq st.1.p$; thus $t_r < et(1,p)$. Between $t_r$ and $et(1,p)$, a complete execution of the acknowledged_reading primitive to read $R_{pq}$ has been done in order to obtain $L_{Reg_{qp}} copy \_state(q) = st.1.p$ at time $et(1,p)$. Therefore, $r$ finishes before time $et(1,p)$.

Theorem 4.7 Let $q$ and $r$ be two neighbours of processor $p$. Let $t_q$ (resp. $t_r$) be a time where a call of $\tau$atomic-state-read by $q$ (resp. $r$) to get p’s state terminates. If $t_r > t_q \geq et(1,p)$ then (i) at time $t_q$, $q$ gets the value st.i.q.p where $i_q \geq 1$, (ii) at time $t_r$, $r$ gets the value st.i.r.p where $i_r \geq 1$, and (iii) $i_r \geq i_q$.

Proof: If we have $t_r > mt(i_q+1,p)$ then $i_r > i_q \geq 1$ otherwise $i_r = i_q \geq 1$ (see Lemma 4.5 and Lemma 4.6).

Lemma 4.8 Only the first $\tau$atomic-state-read of p’s state by $q$ can be a wrong-read.

Proof: A $\tau$atomic-state-read contains a last execution of acknowledged_reading to read $R_{pq}$. Let $t$ be the end time of this execution. We have $mt(1,p) \leq t \leq et(1,p)$ (see lemma 4.6). Between time $t$ and $et(1,p)$, we have $R_{pq} \_local \_state = st.1.p$.

The next wrong-read call should end before $et(1,p)$ (see lemma 4.6). Therefore, its starting time $t'$ is also before $et(1,p)$. At time $t'$, we should have $R_{pq} \_local \_state \neq st.1.p$. This is a contradiction.

Between the time $mt(1,p)$ and the time $et(1,p)$, a call of $\tau$atomic-state-read to get p’s state can return st.1.p, and another call (by another neighbour $q$ of $p$) that terminates after the first one can return another value (i.e. the initial value of $L_{Reg_{qp}} \_local \_state(q)$, the initial value of $R_{pq} \_local \_state$, the initial value of $L_{Reg_{qp}} \_local \_state(p)$, or the initial value of state($p$)).

Complexity The size of each register is $2 \cdot \log(M)+2$ where $M$ is the number of processor states of the algorithm $Alg$ in $A$. The compiled algorithm in the atomic-link model needs only bounded link registers if $Alg$ requires only bounded state registers. An atomic-state-write operation requires at
least \(|A' : p|\) ATOMIC-LINK-READ and ATOMIC-LINK-WRITE operations. But there is no limit on the number of operations performed during an ATOMIC-STATE-READ operation. The duration of the \(\tau(\text{ATOMIC-STATE-WRITE})\) on \(p\) depends on the speed of \(p\)'s neighbor (more precisely, on how often, they read \(p\)'s registers). The \(\tau(\text{ATOMIC-STATE-READ})\) also takes time, a processor may be locked for sometime, before obtaining a neighbor state.

References


