# Self-Stabilizing implementation of atomic register by regular register in network framework

Rapport de Recherche LRI n$^o$ 1449

**Lisa Higham**
Computer Science Departement
University of Calgary
Canada
higham@cpsc.ugalgary.ca

**Colette Johnen**
Laboratoire de Recherche en Informatique
CNRS–Université de Paris-Sud 11
France
colette@lri.fr

**Abstract** In shared-registers communication model for distribued systems, neighbouring processors communicate by reading or writing shared registers. Variants of shared communication register models occur in the literature. In this paper we study the variants where each processor owns a single-writer/multi-reader register. Each register is writable by this owner and readable by each neighbour of the owner. Two comunication models using this type of registers exists, they are determined by selecting the register type (atomic or regular): atomic-statemodel, regular-state model. We present a self-stabilizing compiler to transform an algorithm under the atomic-state model into one under the regular-state model. The presented algorithm does not require the participation of any processor once its out-register has a stabilized content. Moreover the compiler preserves the silent property.
**Keywords:** distributed algorithms, communication models, regular registers, atomic registers, self-stabilization

**Résumé:** Dans ce papier, nous intéressons aux modèles de communication par des registres partagés dans le cadre des système distribué : Les processeurs voisins communique via un registre partagé. Des variantes de ce modèle de communication existent. Dans ce papier, nous étudions les variantes où chaque processor possède un seul registre du type "mono-écrivain, multi-lecteurs", ce registre est lisible par tous les voisins de son propriétaire. Deux modèles existent, ils sont déterminés par le type du registre (atomique ou régulier) : atomic-state modèle et le regular-state modèle.
Nous présentons un compilateur qui transforme un algorithme conçus atomic-state modèle en un algorithme qui s'exécute sur le regular-state modèle. Une fois que le contenu d'un registre est correct, le propriétaire de ce registre peut n'effectuer aucune opération.. Notre compilateur préserve donc la propriété de silence.
**Mots clés:** algorithmes répartis, modèle de comminication, registre régulier, registre atomique, auto-stabilization.

# 1 Introduction

There is a proliferation of models for distributed computing, consisting of both shared memory and message passign paradigms. Different communities adopt different variants as the "standard" model for their research setting. Dolev [4] introduced an 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 [4, 2]. This model can be used to simulate a message passing setting. Let us call this model the *atomic-link* model. 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 [12]. 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.

In the other hand, the consistency condition under concurrent access is a very important characteristic of a register. In this paper, we are interested by two of the three consistency conditions presented by Lamport [10] in increasing order of strength: regular and atomic. Program designs are lot easier with atomic registers than with regular registers but the hardware implementation of an atomic register is costlier than the implemetation of a regular register. This leads us to consider four models determined by selecting the register types (atomic or regular) and the register locations (processors or links).

This paper addressees the differences between state communication models, and the design of fault-tolerant compilers from atomic-state model to regular-state model.

One way of dealing with the large variety of models is to first design for a more abstract or simpler model and then exploit conversion techniques to transform the first solution to one for a more realistic model. 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 compilers that have built-in fault-tolerance. The fault-tolerant model considered in this paper is self-stabilization, which is intended to capture recovery of a distributed system from transient errors of components.

**Related research** Wait free (but not self-stabilizing) transformation from one register model to another one have been extensively studied [1, 5, 11, 13]. Hoepman, Papatrianfafiou and Tsigas [9] 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 globally shared memory model - where any processor can share register with any other processor, i.e. can communicate with any other processor -

Dolev and Herman [3] presented versions of Dijkstra's algorithm that are correct for regular register or safe register, rather than just atomic registers.

In [7], we have established 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 ([13], 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 [7], we also present a self-stabilizing compiler from networks where neighbours communicate via atomic-state registers to systems where neighbours communicate via atomic-link registers. The presented compiler does not preserve the silence property.

**Paper contributions** In section 2, we formally present basic communication models based on shared registers between neighbours. We give the formal definition of a compiler from distributed systems where neighbours communicate via atomic registers to another distributed system where neighbours communicate via regular registers. In section 3, we present a self-stabilizing compiler from distributed systems on network graph where neighbours communicate via atomic-state registers in systems where neighbours comminicate via regular-state registers. In section 5, we give concluding remarks and we prove that the compiled algorithm is silent if the initial algorithm was silent.

# 2 Definitions

## 2.1 Distributed Systems

**Shared registers** Let $\mathcal{R}$ be a single-writer/multi-reader register that can contain any value in domain $T$. $\mathcal{R}$ 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 $\mathcal{R}$ happen sequentially, so they cannot overlap. READ operations, however, may overlap each other and may overlap a WRITE. Lamport [10] defined several kinds of such registers depending of the semantics when READ and WRITE operations overlap. Let $I$ be a set of READ and WRITE operations labelled with their time intervals. Register $\mathcal{R}$ is *regular* if each READ that does not overlap any WRITE returns the value of the most recent preceding WRITE, and any READ that overlaps a WRITE returns either the most recent preceding WRITE or the value of an overlapping WRITE. A sequence of READ and WRITE operation intervals on a regular register is *valid for regular registers* if each READ interval in the sequence satisfies this condition. Register $\mathcal{R}$ is *atomic* if it is regular, and, if a READ overlaps a WRITE and returns the value being written, then any subsequent READ that overlaps the same WRITE must not 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.

**Network models** A distributed network can be modelled by a graph $G = (V, E)$ where $V$ is a set of processors and an edge $\langle pq \rangle \in E$ 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 several variants where each processor uses a collection of local registers

accessible only to itself and communicates with its neighbours via shared registers. The type of register and the way these registers are shared distinguishes the various models.

In the *atomic-state* network model, each processor $p$ owns a single-writer multi-reader shared atomic register $\mathcal{R}_p$, 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:

- ATOMIC-WRITE($\mathcal{R}$, $\nu$) to denote the write of value $\nu$ to the shared register $\mathcal{R}$.
- $\nu \longleftarrow$ATOMIC-READ($\mathcal{R}$) to denote the read of the shared register $\mathcal{R}$ that returns the value $\nu$.

The *regular-state* network model is the same as the atomic-state model except that the shared registers are regular rather than atomic. The WRITE and READ operations
are denoted REGULAR-WRITE($\mathcal{R}$, $\nu$) and $\nu \longleftarrow$REGULAR-READ($\mathcal{R}$) respectively.

**Distributed algorithms, distributed systems** A *distributed algorithm* is an assignment of a program to each processor in the network, and 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. In a *computation step*, several processors simultaneously execute the next step of their programs. A *computation* of a distributed system is a maximal sequences of configurations that are reached by consecutive computation steps. In the atomic-state model and in the regular-state model, one can assume that only one processor performs a move during a computation steps because all compuation steps are serializable.

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

## 2.2 Self-Stabilization

Informally, 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 behavior. Let $\mathcal{P}$ be a predicate defined on configurations. The set of configurations verifying $\mathcal{P}$ is an *attractor* iff

> **convergence:** starting from any configuration, any computation reaches a configuration satisfying $\mathcal{P}$.

**closure:** the reached configuration from any configuration satisfying $\mathcal{P}$ by any computation step satisfies $\mathcal{P}$.

A distributed system is *self-stabilizing to* $\mathcal{L}$ ($\mathcal{L}$ be a predicate defined on configurations) iff

**convergence and closure:** The set of configurations verifying $\mathcal{L}$ is an attractor.

**correctness:** Any computation from a configuration satisfying $\mathcal{L}$ have a correct behavior.

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*.

### 2.3 System transformations and compilers

A transformation of one 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. For example, let $G$ be a graph; we denote by AS($G$) the atomic-state network with topology $G$, and we denote by RS($G$) the regular-state network with topology $G$. To transform an algorithm for AS($G$) to an algorithm for RS($G$) we replace each ATOMIC-WRITE and ATOMIC-READ by every processor $p$ in AS($G$) with a program for $p$ in RS($G$) that uses only local operations and the operations REGULAR-WRITE and REGULAR-READ. Thus a *program transformation* from AS($G$) to RS($G$) is just a mapping. $\tau$(ATOMIC-WRITE($\mathcal{R}$, $\nu$)), and $\tau$(ATOMIC-READ($\mathcal{R}$)) are programs whose operations are on registers in RS($G$) and such that $\tau$(ATOMIC-READ($\mathcal{R}$)) returns a value. Moreover the programs are correct, meaning that during the execution of one of these programs, a processor $p$ does WRITE operation (resp. READ) on register(s) writable by $p$ (resp. readable by $p$).

We are concerned with program transformations from atomic-state model to regular-state model that preserve correctness.

Let $\mathcal{A}$tomic-alg be an program for the atomic state network AS($G$). Let $\tau$ be a program transformation from AS($G$) to thr regular-state network $RS(G)$. A computation $C$ of $\tau(\mathcal{A}\text{tomic-alg})$ on $RS(G)$ is *Linearizable* if the collection of WRITE and READ operations in $C$ correct and valid for atomic registers. A READ (resp. WRITE) operation by the processor $p$ is correct if it contains only REGULAR-READ and REGULAR-WRITE operations that $p$ may perform according the topology $G$. It is straightforward to check that the validity condition is the same as *Linearizability* as used by Lamport [10] and named and used by Herlihy and Wing [6]. That is, for a Linearizable computation, there is a *linearization point* for each exection of $\tau$(o) in $RS(G)$ between the invocation and response of $\tau$(o) such that with these executions ordered according to their linearization point, each $\tau$(ATOMIC-READ) returns the value of the most recent preceding $\tau$(ATOMIC-WRITE) to the same register. The algorithm $\tau(\mathcal{A}\text{tomic-alg})$ *implements* $\mathcal{A}$*tomic-alg on* $RS(G)$ if every computation of $\tau(\mathcal{A}\text{tomic-alg})$ is Linearizable. In this case $\tau(\mathcal{A}\text{tomic-alg})$ is an *implementation* of $\mathcal{A}$tomic-alg on $RS(G)$

A *compiler from atomic-state model on graph* $G$ *to the regular-state network model on the same graph for a class of algorithms* $\mathcal{A}$ is a transformation that implements every $\mathcal{A}lg \in \mathcal{A}$ on the regular-state model on graph $G$. A transformation is a *self-stabilizing compiler* if it is a compiler and it maps self-stabilizing systems to self-stabilizing systems.

# 3 Self-stabilizing Compiler from atomic-state to regular-state

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

every processor executes at least one time $\tau$(ATOMIC-WRITE) procedure after any transient failures.

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

A $\tau$(ATOMIC-WRITE) execution by a processor $p$ has three phases. During the first phase, $p$ selects the first color not used by its neighbours (see the code of the function *setColor* in algorithm 1). The second phase terminates at the end of unsafe section, the first action of this phase is setting $\mathcal{R}_p.Wflag$ to value 0. It is during this phase that $p$ writes the new state and the new color, in its out-register associated with the $Wflag$ value: 0. A $p$'s neighbour that reads the new value associated with the flag 0 cannot used immediatly this new state: it has to wait. The ending time of the second phase is the linearization point of the $\tau$(ATOMIC-WRITE) execution (program counter is at [Wlp]. During the third phase, $p$ sets the $Wflag$ field of its out-register to 1. Now the $p$'s neighbours can used the new state.

During a $\tau$(ATOMIC-READ) execution, if a REGULAR-READ operation returns a state $s$ associated with the flag 1, then the $\tau$(ATOMIC-READ) execution terminates (the returned value is $s$). Otherwise, a $\tau$(ATOMIC-READ) execution is done in two steps. First step, the processor memorizes the value of *state* field ($s$) and *color* field of the readed register. During the second step, the processor does REGULAR-READ operations until *color* field of readed register has changed. The returned value is $s$.

# 4 Algorithm analysis

**Definition 1** *The* REGULAR-WRITE *and* REGULAR-READ *operations are not atomic; they take time to complete. We denote by $t_I(o)$ the invocation time of $o$ and by $t_R(o)$ the response time of $o$. If $o$ is a* REGULAR-WRITE *operation, we name $res(o)$ the record returned. $res(o)$ has three fields : state, color and $Wflag$.*

In self-stabilizing framework, a computation may start in any abritary state. The processors start executing their code at any point. Such computations happen when a transient error has alterated the program counter value. If initially the program counter is inside the $\tau$(ATOMIC-WRITE) (resp. $\tau$(ATOMIC-READ)) code, the processor does a partial $\tau$(ATOMIC-WRITE) (resp. $\tau$(ATOMIC-READ)) partial execution. For each processor, only its first $\tau$(ATOMIC-WRITE) or $\tau$(ATOMIC-READ) execution may be partial.

**Definition 2** *Let $p$ be a processor.*

*$st(i,p)$ denote the start time of the $i$th call of $\tau$(*ATOMIC-WRITE*) by processor $p$.*

*$et(i,p)$ denote the end time of the $i$th call of $\tau$(*ATOMIC-WRITE*) by processor $p$.*

**Algorithm 1** Self-stabilizing compiler from atomic-state systems to regular-state systems

**Constant :**
$B$ a positive integer greater than maximun degree in the network

**Structure of a regular register :**
$\mathcal{R}$= ($state$, $color$, $Wflag$) where $state$ field has state values of the initial algorithm, $color$ takes value in $[1, B]$, and $Wflag$ is a boolean

**Local Variables on p** :
$local\_R$ - local copy of the content of $\mathcal{R}_p$
$\forall q \in \mathcal{N}.p$, ($\mathcal{N}.p$ is the neighbours set of $p$), $local\_R_q$ - local copy of the content of $\mathcal{R}_q$

**Code on the processor p** :
*function setColor()*
*local variable at the function :* $FreeColors$ is a set of colors (i.e. integers)
$FreeColors$ := set of positive integer less or equal to $B$
**For** every neighbour $q$ of $p$ **do**
$local_R_q \longleftarrow$REGULAR-READ($\mathcal{R}_q$);
$FreeColors := FreeColors$ - { $local\_R_q.color$ }
**done**
return min {c such that c $\in FreeColors$};

$\tau$(ATOMIC-WRITE)($\mathcal{R}_p$, new_state)
$local\_R.color$ := setColor(); $local\_R.Wflag := 0$;
[**** begin of the unsafe section ****]
$local\_R.state := new\_state$; REGULAR-WRITE($\mathcal{R}_p$, $local\_R$);
[**** end of the unsafe section ****]
[Wlp] $local\_R.Wflag := 1$; REGULAR-WRITE($\mathcal{R}_p$, $local\_R$);

$\tau$(ATOMIC-READ)($\mathcal{R}_q$)
*local variables at the procedure :* $color$, $state$, and $step$ (a boolean variable)
$step := 0$;
**while** True **do**
$local_R_q \longleftarrow$REGULAR-READ($\mathcal{R}_q$)
if ($local\_R_q.Wflag == 1$) then return $local\_R_q.state$; fi
if ($step == 1$) and ($color \neq local\_R_q.color$) then return $state$; fi
if ($step == 0$) then
$step := 1$; $color := local\_R_q.color$; $state := local\_R_q.state$; fi
$local\_R.color := color$; REGULAR-WRITE($\mathcal{R}_p$, $local\_R$);
**done**

*$mt(i,p)$ denote the time during the ith call of $\tau$(ATOMIC-WRITE) by processor p where code before the label [Wlp] has completed and the code of the following line has not begun to be executed. If the ith call of $\tau$(ATOMIC-WRITE) by processor p does not exist then $mt(i,p)$ has the value $+\infty$.*

*The written state during the ith execution of $\tau$(ATOMIC-WRITE) by p is denoted st.i.p.*

**Definition 3** *Let p be a processor. Let $\mathcal{A}ct$ be a complete $\tau$(ATOMIC-READ) execution by processor p.*

*$st(Act)$ denotes the start time of $\mathcal{A}ct$.*

*$et(Act)$ denotes the end time of $\mathcal{A}ct$.*

### 4.1 Terminaison

In this section, we prove that any execution (partial or complete) of $\tau$(ATOMIC-WRITE) and $\tau$(ATOMIC-READ) terminates.

**Observation 4.1** *Let o an* REGULAR-READ *operation to get the state of p such that $res(o).state = st.i.p$ and $res(O).Wflag = 0$. This operation overlaps the time interval $[st(i,p), et(i,p))$.*

**Lemma 4.2** *Let p be a processor. Any $\tau$(ATOMIC-WRITE) execution by p terminates.*

**Proof:** We name $\delta$ the degree of processor $p$. During the execution of $\tau$(ATOMIC-WRITE), $p$ performs: at most $5 + \delta$ internal operations, two REGULAR-WRITE operations, and $\delta$ REGULAR-READ operations. □

**Lemma 4.3** *Let p be a processor. Any $\tau$(ATOMIC-READ) execution by p terminates.*

**Proof:** Assume that a $\tau$(ATOMIC-READ) execution by $q$ to read $p$ state never ends. Thus, processor $q$ executes infinitely often REGULAR-READ($\mathcal{R}_p$) operation.

**First case -** $p$ executes a finite number of times $\tau$(ATOMIC-WRITE). $\mathcal{R}_p$ will eventually keeps the same value forever. Let us name $t$ the ending time of the last execution of $\tau$(ATOMIC-WRITE) by $p$. At and after $t$, we have $\mathcal{R}_p.Wflag = 1$. Let us name $of$ an REGULAR-READ($\mathcal{R}_p$) operation by $q$ such that $t_I(of) > t$. $of$ is the last REGULAR-READ($\mathcal{R}_p$) operation by $q$ during the $\tau$(ATOMIC-READ) execution because $res(of).Wflag = 1$.

**Second case -** $p$ executes an infinite number of times $\tau$(ATOMIC-WRITE). Let us name $o1$ an REGULAR-READ($\mathcal{R}_p$) operation by $q$. After this operation, $step$ will keep the value 1 forever; $\mathcal{R}_q.color$ and $color(q)$ keep the same value forever. It exists $i$ such that $t_T(o1) < st(i,p)$ because $p$ executes an infinite number of times $\tau$(ATOMIC-WRITE). According the code of $\tau$(ATOMIC-WRITE), at and after $mt(i,p)$ time, we have $\mathcal{R}_p.color \neq color(q)$. Let us name $of$ an REGULAR-READ($\mathcal{R}_p$) operation by $q$ such that $t_I(of) > mt(i,p)$. $of$ is the last REGULAR-READ($\mathcal{R}_p$) operation by $q$ during the $\tau$(ATOMIC-READ) execution because we have $(res(of).color \neq color(q)) \wedge (step = 1)$. □

## 4.2 Convergence

In this section, we will prove the set of configurations verifiying $\mathcal{L}$ is an attractor. First, we define the predicate $\mathcal{L}$.

**Definition 4** *Let $p$ be a processor.*
$Pre\_Correct\_State(p) \equiv [(local\_R_p.Wflag = 0) \vee$ *p's program_counter is not in the unsafe section]*

$\mathcal{C}orrect\_state(p) \equiv Pre\_Correct\_State(p) \wedge$
$[(local\_R_p.state = \mathcal{R}_p.state) \vee$
$((local\_R_p.Wflag = 0) \wedge$ *p's program_counter is not at the begining of line Wlp)].*

$\mathcal{L} \equiv \forall p$, $\mathcal{C}orrect\_state(p)$ *is verified.*

**Lemma 4.4** *Let $p$ be a processor.*
$L1(p) =$*{ configurations such that Pre_Correct_State(p) is verified } is an attractor.*

**Proof:** $p$ does no verify the predicate $Pre\_Correct\_State(p)$ only during its first two steps - in the case where $p$'s program counter was initially in the unsafe section of $\tau$(ATOMIC-WRITE) code. Once the $p$'s program counter is out of the unsafe section. $Pre\_Correct\_State(p)$ is verified.
$Pre\_Correct\_State(p)$ is closed. Because, $p$ sets the $local\_R_p.Wflag$ value to 0 before entering in the unsafe section. □

**Lemma 4.5** *Let $p$ be a processor.*
$L2(p) =$*{ configurations such that $\mathcal{C}orrect\_state(p)$ is verified } is an attractor from any configuration that satisfied $L1(p)$.*

**Proof:** at time $mt(1,p)$, we have $(local\_R_p.state = \mathcal{R}_p.state)$; thus $L2(p)$ is verified.
We will prove that once $L1(p)$ is verified, $L2(p)$ is closed. Assume that $local\_R_p.state$ value is equal to the value of $\mathcal{R}_p$.state. $local\_R_p.state$ value will take a value distinct of $\mathcal{R}_p.state$ value only when $p$'s program counter is inside the unsafe section. But in this case, $((local\_R_p.Wflag = 0) \wedge$ $p$'s program_counter is not at the begining of line Wlp) is verified. $((local\_R_p.Wflag = 0) \wedge$ $p$'s program_counter is not at the begining of line Wlp) stays verified until $p$ executes the last action of unsafe section: REGULAR-WRITE($\mathcal{R}_p$, $local\_R$). After this action, we have $(local\_R_p.state = \mathcal{R}_p.state)$; thus $\mathcal{C}orrect\_state(p)$ is verified. □

## 4.3 Linearization Points

In this section, we define for any $\tau$(ATOMIC-WRITE) and $\tau$(ATOMIC-READ) execution their linearization point.

### 4.3.1 Linearization Points after $mt(1,p)$

**Observation 4.6** *Let $i$ be an integer greater than 0.*
*Let $o_b$ an* REGULAR-READ *operation to get the state of $p$ such that $t_R(o_b) < mt(i,p)$. We have:* $(res(o_b).state \neq st.i.p) \vee (res(o_b).Wflag = 0)$.

*Let $o_a$ an* REGULAR-READ *operation to get the state of $p$ such that $t_I(o_a) \geq mt(i+1,p)$. We have: $res(o_a).state \neq st.i.p$.*

**Lemma 4.7** *Let $\mathcal{A}ct$ be a complete $\tau$(*ATOMIC-READ*) execution by $q$ to get the atomic state of $p$. Assume that the returned value of $\mathcal{A}ct$ is $st.i.p$ with $i \geq 1$. The time interval of $\mathcal{A}ct$ execution overlaps with $[mt(i,p),\ mt(i+1,p))$.*

**Proof:** The returned value (named $res(Act)$) is the field *state* of $res(o)$ where $o$ is one of the REGULAR-READ operation performed by $q$ during the execution $\mathcal{A}ct$.

**First case :** $res(o).Wflag = 1$. From the Observation 4.6, we conclude that if the returned value of $\mathcal{A}ct$ is $st.i.p$ with $i \geq 1$, then the time interval of $\mathcal{A}ct$ overlaps with $[mt(i,p),\ mt(i+1,p))$.

**Second case :** $res(o).Wflag = 0$. Notice that $o$ is the first REGULAR-READ($\mathcal{R}_p$) operation in $\mathcal{A}ct$. We have $res(o) = (st.i.p, c, 0)$, thus the execution of $\mathcal{A}ct$ starts before $mt(i,p)$, according to Observation 4.6. Later, during the execution of $\mathcal{A}ct$, $q$ performs a REGULAR-READ operations $o'$ such that $res(o') = (s, c', 0)$ with $c \neq c'$. We conclude that $t_I(o') > st(i',p)$ where $i' > i$, according to the $\tau$(ATOMIC-WRITE) code. Thus $\mathcal{A}ct$ starts before $mt(i,p)$ and ends after $st(i',p)$. Therefore $\mathcal{A}ct$ overlaps the time interval $[mt(i,p),\ mt(i+1,p))$. □

**Lemma 4.8** *Let $ct$ be a complete $\tau$(*ATOMIC-READ*) execution by $q$ to get the atomic state of $p$. Assume that the returned value of $\mathcal{A}ct$ is $s$ and it does not exist $i > 0$ such that $s = st.i.p$. The time interval of $\mathcal{A}ct$ execution overlaps with $[0,\ mt(1,p))$.*

**Proof:** The returned value (named $res(Act)$) is the field *state* of $res(o)$ where $o$ is one of the REGULAR-READ operation performed by $q$ during the execution $\mathcal{A}ct$.

Let $o$ an REGULAR-READ operation such that $t_I(o) \geq mt(1,p)$. It exists $i > 0$ such that $res(o).state = st.i.p$. □

**Definition 5** L*inearization points after $mt(1,p)$*
*The linearization point of the ith call of $\tau$(*ATOMIC-WRITE*) by $p$ is $mt(i,p)$.*

*Let $\mathcal{A}ct$ be a complete $\tau$(*ATOMIC-READ*) execution by $q$ to get the atomic state of $p$. If the returned value of $\mathcal{A}ct$ is $st.i.p$ where $i \geq 1$ then the linearization point of $\mathcal{A}ct$ is fixed to any time during the interval $[mt(i,p),\ mt(i+1,p)) \cap [st(Act),\ et(Act)]$.*

### 4.3.2 Linearization Points before $mt(1,p)$

**Definition 6** *We call $lt(p)$ the the first time point where $\mathcal{C}orrect\_state(p)$ is verified.*

**Observation 4.9** *We have $lt(p) \leq mt(1,p)$.*

Now, we study the behavior of the $p$ before $mt(1,p)$.

**Observation 4.10** *Before the time $mt(1,p)$, the field $\mathcal{R}_p$.state may have two consecutively distinct values: its initial values, and then the value that was initially in local_R.state(p), or the initial value of new_state(p).*
*The partial $\tau$(ATOMIC-WRITE) by p contains only one* REGULAR-WRITE *operation where the written $Wflag$ value is 0.*

**Definition 7** *We denote by st.-1.p the value of the field $\mathcal{R}_p$.state at time 0.*
*We denote by st.0.p the state value written during the only partial execution of $\tau$(ATOMIC-WRITE).*
*We denoted by $mt(0,p)$ the ending of the* REGULAR-WRITE *operation where st.0.p was written for the first time in $\mathcal{R}_p$.*
*If the partial execution of $\tau$(ATOMIC-WRITE) does not exist then we denote by st.0.p the value of st.-1.p and $mt(0,p)=0$.*

Once $L2(p)$ is reached, an execution of REGULAR-WRITE by $p$ that changes the value of field $\mathcal{R}_p.state$ is done where $local\_R_p.Wflag = 0$. If this operation is done during the ith call of $\tau$(ATOMIC-WRITE) by processor $p$, the ending time of the operation is $mt(i,p)$. If this operation is done during a partial execution of $\tau$(ATOMIC-WRITE) this time is $mt(0,p)$.

**Observation 4.11** *Let $o_a$ an* REGULAR-READ *operation to get the state of p such that $t_I(o_a) \geq mt(1,p)$. We have: $res(o_a).state \neq st.\text{-}1.p$ and $res(o_a).state \neq st.0.p$.*
*Assume that $lt(p) < mt(0,p)$. Let $o_b$ an* REGULAR-READ *operation to get the state of p such that $LTp < t_R(o_b) < mt(0,p)$. We have: $(res(o_b).state = st.\text{-}1.p) \vee (res(o_b).Wflag = 0)$.*

The proof of lemma 4.12 is similar to the proofs of lemma 4.7 and 4.8.

**Lemma 4.12** *Let $\mathcal{A}ct$ be a complete $\tau$(ATOMIC-READ) execution by q to get the atomic state of p such that $st(\mathcal{A}ct) \geq lt(p)$.*
*Assume that the returned value of $\mathcal{A}ct$ is st.0.p. The time interval of $\mathcal{A}ct$ execution overlaps with [SUP(lt(p),mt(0,p)), mt(1,p)).*
*Assume that the returned value of $\mathcal{A}ct$ is st.-1.p. We have SUP(lt(p),mt(0,p)) = mt(0,p) and the time interval of $\mathcal{A}ct$ execution overlaps with [lt(p), mt(0,p)).*

**Definition 8** L*inearization points before $mt(1,p)$*
*The linearization point of the partial execution of $\tau$(ATOMIC-WRITE) by p is $mt(0,p)$.*

*Let $\mathcal{A}ct$ be a complete $\tau$(ATOMIC-READ) execution by q to get the atomic state of p. If $st(\mathcal{A}ct) < lt(p)$ then the linearization point of $\mathcal{A}ct$ is fixed to any time during the interval [0, lt(p)].*

*Let $\mathcal{A}ct$ be a complete $\tau$(ATOMIC-READ) execution by q to get the atomic state of p such that $st(\mathcal{A}ct) \geq lt(p)$. If the returned value of $\mathcal{A}ct$ is st.0.p then the linearization point of $\mathcal{A}ct$ is fixed to any time during the interval [SUP(lt(p),mt(0,p)), mt(1,p)) $\cap$ [st(Act), et(Act)].*
*If the returned value of $\mathcal{A}ct$ is $s = st\text{-}1.p$, then the linearization point of $\mathcal{A}ct$ is fixed to any time during the interval [lt(p), mt(0,p)) $\cap$ [st(Act), et(Act)].*

### 4.4 Correctness

In this section, we prove that the linearization points occuring after $\mathcal{L}$ verifying are valid.

**Theorem 4.13** *Let $p$ be a processor. All complete $\tau$(ATOMIC-READ) of $p$ state and $\tau$(ATOMIC-WRITE) by $p$ such that their linearization points are after $lt(p)$ are valid for atomic registers.*

**Proof:** All $\tau$(ATOMIC-READ) of $p$ state and $\tau$(ATOMIC-WRITE) by $p$ such that their linearization points are after $lt(p)$ verify the following properties:

- The linearization point of $\mathcal{A}ct$ (a ATOMIC-READ, or ATOMIC-WRITE operation) is between the invocation and response of $\tau(\mathcal{A}ct)$ - see lemma 4.7, 4.8 and 4.12.
- Once the $\tau$(ATOMIC-READ) and $\tau$(ATOMIC-WRITE) executions are ordered according to their linearization point, each $\tau$(ATOMIC-READ) of $p$'s state returns the value of the most recent preceding $\tau$(ATOMIC-WRITE) by $p$ - according to their linearization points definition. see definition 5, and 8.

□

## 5 Conclusion

We have presented a compiler from atomic-state model to regular-state model. We have proven that any partial or complete execution of $\tau$(ATOMIC-READ) and of $\tau$(ATOMIC-WRITE) terminates. Our compiler is self-stabilizing: once a legitimate configuration is reached, the executions of $\tau$(ATOMIC-READ) and of $\tau$(ATOMIC-WRITE) program have a valid linearization point whatever the performed computation.

**Memory Space Complexity** The size of each register needs by the compiled algorithm is $log(M) + 1 + B$ bits where $M$ is the number of processor states of the algorithm $\mathcal{A}lg$ in $\mathcal{A}$. The compiled algorithm in regular-state model needs only bounded registers if $\mathcal{A}lg$ requires only bounded state registers.

**Wait-freedom** Informally, an operation is wait-free if no processor invoking the operation can be forced to wait indefinitely for another processor. Such robustness implies that a stopping failure (or very slow execution) of any subset of processors cannot prevent another processor from correctly completing its operation. Formally an 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. $\tau$(ATOMIC-WRITE) is a wait free operation. $\tau$(ATOMIC-READ) is not a wait free operation.
Using a wait free compiler from atomic-state models to regular-state model, one can design a wait free compiler from atomic-state models to atomic-link model (see [8]). In [7], we have proven that there is not wait-free compiler from atomic-state model to atomic-link model. Thus, it does not exist a wait free compiler from atomic-state model to regular-state model.

**Silence** Notice that the compiler preserves the silence property. An algorithm is silence if once a stable configuration is reached processors will never change their state. Processors only check if their neighbor states has changed or not. The silence property is desirable property in terms of simplicity and communication overhead. Assume that the intial algorithm $\mathcal{A}lg$ is silent. After the stabilization phase, (i) processors will only execute $\tau$(ATOMIC-READ) code; (ii) on any register $\mathcal{R}$, we have $\mathcal{R}.Wflag = 1$. Thus, a $\tau$(ATOMIC-READ) execution does not require a REGULAR-WRITE operation, once $\mathcal{A}lg$ is stabilized. Therefore, $\tau(alg)$ is silent if $\mathcal{A}lg$ is silent. Thus the presented compiler preserves the silence property.

## References


[1] Uri Abraham. On interprocess communication and the implementation of multi-writer atomic registers. *Theoretical Computer Science*, 149(2):257–298, 1995.

[2] S Dolev. *Self-Stabilization*. MIT Press, 2000.

[3] S Dolev and T Herman. Dijkstra's self-stabilizing algorithm in unsupportive environments. In *WSS01 Proceedings of the Fifth International Workshop on Self-Stabilizing Systems, Springer LNCS:2194*, pages 67–81, 2001.

[4] S Dolev, A Israeli, and S Moran. Self-stabilization of dynamic systems assuming only read/write atomicity. *Distributed Computing*, 7(1):3–16, 1993.

[5] S. Haldar and K. Vidyasankar. Constructing 1-writer multireader multivalued atomic variables from regular variables. *J. ACM*, 42(1):186–203, 1995.

[6] MP Herlihy and JM Wing. Linearizability: a correctness condition for concurrent objects. *ACM Trans. Program. Lang. Syst.*, 12(3):463–492, 1990.

[7] L. Higham and C. Johnen. "relationships between communication models in networks using atomic registers". In *IPDPS'2006 Proceedings of the 20th IEEE International Parallel & Distributed Processing Symposium*, 2006.

[8] L. Higham and C Johnen. Relationships between communication register models in networks. Technical Report 1419, L.R.I, 2006.

[9] JH Hoepman, M Papatriantafilou, and P Tsigas. Self-stabilization of wait-free shared memory objects. *Journal of Parallel and Distributed Computing*, 62(5):818–842, 2002.

[10] L Lamport. On interprocess communication. *Distributed Computing*, 1(2):77–101, 1986.

[11] Ming Li, John Tromp, and Paul M. B. Vitányi. How to share concurrent wait-free variables. *J. ACM*, 43(4):723–746, 1996.

[12] M Nesterenko and A Arora. Stabilization-preserving atomicity refinement. *Journal of Parallel and Distributed Computing*, 62(5):766–791, 2002.

[13] JL Welch and H Attiya. *Distributed computing: fundamentals, simulations and advanced topics*. McGraw-Hill, Inc., 1998.