Relationships between communication register models in networks

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
France
colette@lri.fr

Abstract

One common way to model a distributed system is with 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.
This paper defined four models of shared registers determined by selecting the register types (atomic or regular) and the register locations (processors or links).
We determine under what conditions and with what robustness and/or failure-tolerance guarantees it is possible to transform a solution under one model into a solution under another model.

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

Résumé

Un système réparti est souvent modélisé par un graphe où les noeuds représentent les processeurs. Il existe un arc entre deux noeuds si et seulement si les deux processeurs peuvent communiquer directement (ces processeurs sont dit voisins). Dans ce papier, nous intéressons aux modèles de communication par des registres partagés. Nous définissons quatre modèles de registres partagés. Ces modèles sont déterminés par le type du registre (atomique ou régulier) et par la localisation du registre (processeur ou lien de communication).
Nous déterminons sous quelles conditions il est possible de transformer un algorithme pour un de ces quatre modèles en un algorithme pour un autre modèle. Nous nous intéressons aux compilateurs qui sont tolérant aux pannes à savoir auto-stabilisant et sans-attente (c-a-d wait-free).

mots clés: algorithme réparti, modèles de communication, registre partagé, registre régulier, registre atomique, mono-lecteur, multi-lecteurs, sans-attente auto-stabilisation.

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. In the first paper on Self-stabilizing distributed algorithms [4], 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 [7] introduced an read/write atomicity model for self-stabilizing algorithms to better capture the actual possible communication between processors. Many subsequent papers have used such a model. 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 are, however, two variants of read/write atomicity assumed in the self-stabilization literature. For each link between two processors, there are two single-writer/single-reader atomic registers, each one writable by one processors and readable by the other. In several other papers, the read/write model assumes that a single-writer/multi-reader atomic registers reside at each processor. In either of these two 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 [18] 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).

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. This paper addressees the differences between these communication models, and the design of fault-tolerant compilers from one model to another. 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 models considered in this paper are wait-freedom, which is intended to capture tolerance of stopping failures of components of a distributed system, and self-stabilization, which is intended to capture recovery of a distributed system from transient errors of components.

Paper contributions In section 2, we formally present the four basic communication models based on shared registers between neighbours. We also formally present the both fault-tolerant models that we are considering in this paper.We five the formal definition of a compiler from distributed systems where neighbours communicate via a type of register to another distributed system where neighbours communicate via another type of register. After having defined the communication registers models, we can do an accurate presentation of the related works, in section 3.

In the second part of the paper, we study the relationship between two register models into network framwork: the atomic-state model and the atomic-link model. These models are the most used by the self-stabilizing communities. In section 5, we proof that a wait-free compiler from atomic-state systems to atomic-link systems, requires shared registers between any pair of neighbours of the same processor. In a distributed system based on a network graph, this is not always possible. Two neighbours of a processor may or may not be neighbour. If they are not neighbour then they cannot share any register. In section 6, 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 atomic-link registers.
To conclude, in section 7, we study implementation from one to another basic network model, we discute if wait-free compilers exists or not. For any model to another one, we present a self-stabilizing compiler.

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

In the *atomic-link* network model, for each edge $\langle pq \rangle \in E$, there are two single-writer, single-reader atomic registers. Register $\mathcal{R}_{pq}$ is writable by p and readable by q; register $\mathcal{R}_{qp}$ 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. In an atomic-link network model, the WRITE and READ operations are denoted ATOMIC-LINK-WRITE($\mathcal{R}$, ν) and $\nu \longleftarrow$ATOMIC-LINK-READ($\mathcal{R}$) respectively.

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-STATE-WRITE($\mathcal{R}$, ν) and $\nu \longleftarrow$REGULAR-STATE-READ($\mathcal{R}$) respectively.

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

The four models: atomic-state, atomic-link, regular-state, and regular-link network models are all called *basic* models and are summarized in the Table 1.

	atomic models	*regular models*
state models	atomic-state model	regular-state model
link models	atomic-link model	regular-link model

Table 1: the four basic models

The *composite state* distributed system model is often considered in the self-stabilization literature[4]. In a composite state distributed system, a processor p's step is denoted by a collection of rules of the form: `if P then` WRITE($\mathcal{R}_p$, ν), where P is a predicate on the state of p and the values in the shared registers of each of p's neighbours. Processor p is *enabled* in a given configuration, if at least one of its predicates evaluate to `true`, in which case, in its next step, it would update a shared register. Otherwise p is *disabled*.

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.

Schedulers and computations A *configuration* of a distributed system is a collection of values assigned to all the registers of the system. Informally, computations of distributed systems are the sequences of configurations that are produced as processors execute steps of their algorithms. 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 $\mathcal{C} = C_0, C_1, \ldots$ 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. An *arbitrary* Scheduler has no requirement to eventually select each processor. In a composite state model, the most permissive Scheduler is the *unfair Scheduler*, which is only required to select enabled processors at every step. In a basic model, the atomic steps are reads and writes of registers; hence each processor that has not terminated is always enabled, making it unnecessary to distinguish between processors and enabled processors.

The *centralized Scheduler* restricts each S_i to have size one. The *distributed Scheduler* has no restrictions in the size of the set of processors taking the next step. For any schedule S, each set S_i in S can be any non-empty subset of processors (basic models) or of enabled processors (composite state model). A Scheduler is *fair* if, in every infinite computation, every processor or every processor enabled infinitely often (composite state model) 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 Scheduler $\mathcal{S}$ and network class $\mathcal{N}$* if for any network $N \in \mathcal{N}$ and for any schedule $s \in \mathcal{S}$ the computation of algorithm A on N under schedule s satisfies predicate P.

2.2 Fault-tolerance

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. 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.

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 configurations. Let $\mathcal{L}$ be a predicate defined on configurations. A distributed system is *self-stabilizing to $\mathcal{L}$ under Scheduler $\mathcal{S}$* if, for any schedule $s \in \mathcal{S}$,

convergence: starting from any configuration, any computation under schedule s reaches a configuration satisfying $\mathcal{L}$.

closure: For any configuration C satisfying $\mathcal{L}$ and for any subset S of processors allowed by s, $S(C)$ 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.

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; denote by AS(G) the atomic-state network with topology G, and denote by AL(G) the atomic-link network with topology G. To transform an algorithm for AS(G) to an algorithm for AL(G) we replace each ATOMIC-STATE-WRITE and ATOMIC-STATE-READ by every processor p in AS(G) with a program for p in AL(G) that uses only local operations and the operations ATOMIC-LINK-WRITE and ATOMIC-LINK-READ. Thus a *program transformation* from AS(G) to AL(G) is just a mapping where τ(ATOMIC-STATE-WRITE($\mathcal{R}$, ν)), and τ(ATOMIC-STATE-READ($\mathcal{R}$)) are programs whose operations are on registers in AL(G) and such that τ(ATOMIC-STATE-READ($\mathcal{R}$)) returns a value.

We are concerned with program transformations between basic models that preserve correctness.

Let $\mathcal{A}$tomic-alg be an program for an atomic network AN(G). Let τ be a program transformation from AN(G) to another basic network, $N(G)$. A computation C of $\tau(\mathcal{A}$tomic-alg) on $N(G)$ is *Linearizable* if the collection of operation in C are valid for atomic registers. It is straightforward to check that this correctness condition is the same as *Linearizability* as used by Lamport [18] and named and used by Herlihy and Wing [12]. That is, for a Linearizable computation, there is a *linearization point* for each WRITE and READ operation o in $N(G)$ between the invocation and response of τ(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 $\tau(\mathcal{A}$tomic-alg) *implements* $\mathcal{A}$*tomic-alg on* $N(G)$ if every computation of $\tau(\mathcal{A}$tomic-alg) is Linearizable. In this case $\tau(\mathcal{A}$tomic-alg) is an *implementation* of $\mathcal{A}$tomic-alg on $N(G)$

Now consider a program $\mathcal{R}$egular-alg for a regular network, RN(G), and let γ be a program transformation from RN(G) to another basic network, $N(G)$. A computation C of $\gamma(\mathcal{R}$egular-alg) on $N(G)$ is *Regularizable* if the collection of operations in C are valid for regular registers. The algorithm $\gamma(\mathcal{R}$egular-alg) *implements* $\mathcal{R}$*egular-alg on* $N(G)$ if every computation of $\gamma(\mathcal{R}$egular-alg) is Regularizable. In this case $\gamma(\mathcal{R}$egular-alg) is an *implementation* of $\mathcal{R}$egular-alg on $N(G)$.

A *compiler from a network model* N *on graph* G *to a network model* $\hat{N}$ *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 network model $\hat{N}$ 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. A compiler is *wait-free* if it maps wait-free systems to wait-free systems.

3 Related Research

There are several papers [20, 1, 14, 22] that provide self-stabilizing compilers from composite systems to state systems for various sets of network topologies. Mizuno and Nesterenko's compiler [20] and Antonoiu and Srimani's compiler [1] apply to general topologies where processors have distinct identifiers. These depend on unbounded timestamps. Antonoiu and Srimani also present a self-stabilizing compiler for any spanning tree network that uses bounded timestamps [1]. To ensure safety, no processor can enter the critical section while the timestamps are wrapping around. Huang's compiler [14] applies to anonymous acyclic graphs and uses bounded memory space. Nesterenko and Arora's compiler [22] is based on a bounded space self-stabilizing dining philosophers protocol for state register systems.

Any self-stabilizing Mutual Exclusion protocol or self-stabilizing token circulation protocol under the distributed Scheduler is also a compiler from a composite system under the centralized Scheduler to a composite system under the distributed Scheduler, since they ensure that only one processor is enabled at a time. More efficient solutions [9, 10, 17, 23, 2] are based on Local Mutual Exclusion protocols, which ensure that only one processor in a neighbourhood is enabled. All these transformers require distinct identifiers or use randomization to break symmetry.

Dolev [5] presents several techniques for converting a self-stabilizing protocol from one system to another one. For instance, in networks with distinct identifiers, to get a self-stabilizing compiler from composite systems to atomic-link systems, one can fairly compose Dolev's [5] self-stabilizing Leader Election in atomic-link systems with Dolev *et.al.* [7] self-stabilizing Mutual Exclusion.

They achieve synchronization after $O(n^3)$ steps ([8]), $O(n^2)$ steps ([24]) or $O(n)$ steps ([21]). Hoepman, Papatrianfafiou and Tsigas in [13] 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 into 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 - transformation from on register model to another one have been extensively studied [11, 19, 27].

Dolev and Herman [6] presented versions of Dijkstra's algorithm that are correct for regular register or safe register, rather than just atomic registers.

4 Some Relationships Between Models

Distributed versus centralized schedules

A distributed schedule acting on a basic network could have several processors simultaneously taking their next step, and hence possibly have a write and one or more read operations apply to the same register. However, since the registers are at least regular, the outcome of this step guarantees that each read returns the value of either an overlapping write or the most recent preceding write. Thus the reads and writes of each step could be serialized into a sequence of steps each by only one

processor, so that the outcome is unchanged. In contrast, it is not necessarily possible to serialize the simultaneous composite steps in composite state networks. Under a distributed schedule, two adjacent processors could take a simultaneous step that examines each other's current register value and writes into its own shared register an updated value that is a function of the values read. While there is a ordering of reads and writes that can mimic this result there is no serialization of the composite steps.
Since the simultaneous steps can be serialized if and only if they can be simulated by a centralized schedule, we have our first observation:

Observation 4.1 *For any basic system, correctness under the centralized Scheduler implies correctness under the distributed Scheduler. For a composite state system this does not hold.*

Schedulers, wait-freedom, and self-stabilization

Schedulers can be used to describe wait-freedom. Recall that in the four basic network models, every processor is always enabled, so 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. This is not true in the composite state model because the unfair Scheduler is restricted to choosing from only enabled processors.
A self-stabilizing algorithm under the unfair Scheduler is typically designed so that the unfair Scheduler is forced to eventually choose some processors that were previously blocked because they are the only remaining enabled ones. In this case the algorithm could be self-stabilizing under the unfair Scheduler, but it is not wait-free. Rather, the unfair Scheduler in this composite state model is more closely related to the weakly fair scheduler in the general distributed computing literature, because the unfair Scheduler is forced to act as if it is weakly fair.

Observation 4.2 *For any basic system, self-stabilization under the unfair Scheduler implies wait-free self-stabilization under the unfair Scheduler. For a composite state system this does not hold.*

Safety and liveness for self-stabilizing and wait-free systems

Some further relationships are exposed by examining the safety and liveness requirements of the two major 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.

Relationships between the basic systems

We wish to implement algorithms designed for one basic network on another basic network for the combinations show in Table 1. Our first goal is to examine transforming from atomic-state systems to atomic-link systems. In the next two sections, we show that there is not wait-free compiler and we present a self-stabilizing compiler.

5 Wait-free Compiler from atomic-state to atomic-link

Let G be any connected graph. Given an algorithm $\mathcal{A}lg$ for an atomic-state network AS(G), we would like to implement it on the atomic-link network AL(G). Attiya and Welch ([27] page 366) 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 [3, 15, 25, 26] 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 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 [27] (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 Lamport [18], 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. However, there are now writes occurring by the readers as well as the writers, 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 5.1 *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. As READ of $\mathcal{R}$ is a wait-free operation, it can be performed into 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 $\mathcal{R}$ 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 $\mathcal{R}$ 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 5.2 *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 ATOMIC-STATE-WRITE($\mathcal{R}_w$, v) and ATOMIC-STATE-READ($\mathcal{R}_w$) of a single-writer/multi-reader register $\mathcal{R}_w$ 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 ATOMIC-STATE-WRITE and ATOMIC-STATE-READ operations into programs that use the ATOMIC-LINK-READ and 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 $\mathcal{R}_w$ using single-reader registers. By Lemma 5.1, 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 atomic-state model for G to the atomic-link platform with topology G. □

6 Self-stabilizing Compiler from atomic-state to atomic-link

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

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

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

We show that Algorithm 6 is a self-stabilizing compiler from atomic-state networks to atomic-link networks for all algorithms in $\mathcal{A}$.

The self-stabilizing communication primitives *acknowledged_writing* and *acknowledged_reading* for the atomic-link model appeared earlier [16]. 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 $\mathcal{R}_{qp}$ has $2.k$ fields where k is the number of communication variables. Two fields called $local\_x$ and $copy\_x$ are associated with each communication variable, x. The $local\_x$ field contains the value of variable x that q wants to communicate to p. The $copy\_x$ field contains the last read value of p's variable x by q. During a reading operation by p of register $\mathcal{R}_{qp}$, p copies the values of all $local\_$ fields of $\mathcal{R}_{qp}$ into the $copy\_$ fields of the register $\mathcal{R}_{pq}$. After a writing operation, p checks to determine if the value of each $copy\_$ field of register $\mathcal{R}_{qp}$ 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 $ok.q$ is set to 1. Once all p's neighbours have read the new values of communication variables the *acknowledged_writing* by p is over. Observe that *acknowledged_reading* is not blocking. The following Claim is proved in earlier work [16].

Claim 6.1 ([16]) *Assuming that each processor performs acknowledged_reading infinitely often, any execution of acknowledged_writing eventually completes.*

Algorithm 6.1 Self-stabilizing compiler from atomic-state systems to atomic-link systems

structure of a register :

$\mathcal{R}$= ($local\_state$, $local\_flag$, $copy\_state$, $copy\_flag$) 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
$\forall r \in \mathcal{N}.p$, ($\mathcal{N}.p$ is the neighbours set of p),
ok_r - boolean variable
$Local\_Reg_{pr}$ and $Local\_Reg_{pr}$ - same structure as $\mathcal{R}$

code on the processor p :

τ(ATOMIC-STATE-WRITE)($\mathcal{R}_p$,new_state)
$state := new\_state$;
$flag := 0$; acknowledged_writing(state) [l1]
$flag := 1$; acknowledged_writing(state) [l2]

τ(ATOMIC-STATE-READ)($\mathcal{R}_q$)
repeat
for $r \in \mathcal{N}.p$ **do** $acknowledged\_reading(\mathcal{R}_{rp})$ **done**
until $Local\_Reg_{qp}.local\_flag = 1$ T
return $Local\_Reg_{qp}.local\_state$

$acknowledged\_writing(state)$:
for $r \in \mathcal{N}.p$ **do** $acknowledged\_reading(\mathcal{R}_{rp})$; $ok.r := 0$ **done**
repeat
for $r \in \mathcal{N}.p$ **do**
$acknowledged\_reading(\mathcal{R}_{rp})$
if ($Local\_Reg_{rp}.copy\_state = state$) $\wedge$ ($Local\_Reg_{rp}.copy\_flag = flag$) **then**
$ok.r := 1$
done
until ($\forall r \in \mathcal{N}.p$, $ok.r = 1$)

$acknowledged\_reading(\mathcal{R}_{rp})$:
$Local\_Reg_{rp} \longleftarrow$ ATOMIC-LINK-READ($\mathcal{R}_{rp}$)
$Local\_Reg_{pr}.local\_state = state$; $Local\_Reg_{pr}.local\_flag := flag$;
$Local\_Reg_{pr}.copy\_state := Local\_Reg_{rp}.local\_state$;
$Local\_Reg_{pr}.copy\_flag := Local\_Reg_{rp}.local\_flag$;
ATOMIC-LINK-WRITE($\mathcal{R}_{pr}$,$Local\_Reg_{pr}$);

The communication variable for Algorithm 6 are, for each processor, the state variables (called *state*) used in the algorithm A, plus a flag value (called $flag$).
During the second complete execution of the *acknowledged_writing* by p with distinct flags, all its neighbours perform an ATOMIC-LINK-WRITE operation. This operation may be not inside a complete execution of the *acknowledged_reading* primitive. During the third complete execution of the *acknowledged_writing* by p with distinct flag, all its neighbours perform a ATOMIC-LINK-WRITE operation inside a complete execution of the *acknowledged_reading* primitive. Thus, at the end of third complete executions of the *acknowledged_writing* primitive by p with distinct flags, for any neighbour q of p, $local\_Reg_{qp}.copy\_state(q) = local\_Reg_{pq}.local\_state(q) = state(p)$ and $local\_Reg_{qp}.copy\_flag(q) = local\_Reg_{pq}.local\_flag(q) = flag(p)$ (see [16] for a formal proof).

Lemma 6.2 *For any algorithm $\mathcal{A}lg$ in $\mathcal{A}$, any execution of τ(ATOMIC-STATE-WRITE) by any processor p eventually terminates.*

Proof: The lemma follows immediately from the code for τ(ATOMIC-STATE-WRITE) and Claim 6.1 and the properties of $\mathcal{A}$. □

Definition 1 *Consider the ith call of τ(ATOMIC-STATE-WRITE) by processor p. Let*
$st(i,p)$ denote the start time,
$et(i,p)$ denote the end time,
$mt(i,p)$ denote the time that line [l1] has completed and line [l2] has not begun
The value of state.p during the ith call of τ(ATOMIC-STATE-WRITE) by p is denoted $st.i.p$.

Observation 6.3 *At time $mt(1,p)$, for any neighbour q of p, we have :*
$\mathcal{R}_{pq}.local\_state = state(p) = st.1.p$ *and* $\mathcal{R}_{pq}.local\_flag = flag(p) = 0$.
At time $et(1,p)$ for any p's neighbour q, we have :
$Local\_Reg_{qp}.copy\_state(q) = state(p) = st.1.p$ *and* $Local\_Reg_{qp}.copy\_flag(q) = flag(p) = 1$.
At time $mt(i,p)$, and $et(i,p)$, for $i \geq 2$, for any neighbour q of p, we have :
$Local\_Reg_{qp}.copy\_state(q) = Local\_Reg_{pq}.local\_state(q) = state(p) = st.i.p$ *and*
$Local\_Reg_{qp}.copy\_flag(q) = Local\_Reg_{pq}.local\_flag(q) = flag(p)$.

Lemma 6.4 *Any execution of the τ(ATOMIC-STATE-READ) eventually terminates.*

Proof: Let p and q be two neighbour processors. If q executes τ(ATOMIC-STATE-READ) a finite number of times, then $Local\_Reg_{pq}.copy\_flag(p)$, $Local\_Reg_{qp}.local\_flag(p)$, and $flag(q)$ will eventually keep the value 1 forever. After that time, the execution of τ(ATOMIC-STATE-READ) by p consists of $|\mathcal{N}.p|$ atomic operations ($|\mathcal{N}.p|$ read operations). Therefore, any execution of τ(ATOMIC-STATE-READ) eventually terminates.
Assume that q executes τ(ATOMIC-STATE-WRITE) infinitely often. Let t' be the starting time of an execution of τ(ATOMIC-STATE-READ) by p. Let us call t the next starting time of the execution by q of τ(ATOMIC-STATE-WRITE) after t'. Without lost of generality, we can assume that was the ith call of τ(ATOMIC-STATE-WRITE) by q.

Assume that $i > 1$. 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 $Local\_Reg_{pq}.copy\_flag(q) = 1$. 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 $Local\_Reg_{pq}.copy\_flag(q) = 0$.
Between the time interval $[mt(i,p), mt(i+1,q)]$, p performed at least one time the test T at the time when $Local\_Reg_{qp}.local\_flag(p) = 1$. - between the two executions of the primitive $acknowledged\_reading(Reg_{qp})$ -. Thus, the execution of τ(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 τ(ATOMIC-STATE-WRITE) by p is the time $mt(i,p)$ (where $i > 1$). The linearization point of a τ(ATOMIC-STATE-READ) is its ending time. According to theorem 6.7, each τ(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 τ(ATOMIC-STATE-WRITE) by p.

Lemma 6.5 *Let q and p be two processors neighbour. Let $i > 1$. Let t be a time where a call of τ(ATOMIC-STATE-READ) by q terminates. If $mt(i,p) < t < mt(i+1,p)$ then τ(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 : $(Local\_Reg_{pq}.local\_state(q) = st.i.p \vee Local\_Reg_{pq}.local\_flag(q) = 0)$. 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 neighbour processors.*
We name WRONG-READ *a call of τ(ATOMIC-STATE-READ) that (i) does not return $st.1.p$ and (ii) that terminates during the time interval $[mt(1,p),\ mt(2,p)]$.*

Lemma 6.6 *A* WRONG-READ *terminates before the time $et(1,p)$.*

Proof: At the end of *acknowledged_reading* primitive execution to read $\mathcal{R}_{pq}$ terminating after the time $et(1,p)$, we have $Local\_Reg_{qp}.copy\_state(q) = st.i.p$. where $i \geq 1$ - see Observation 6.3.

Let r be an WRONG-READ of p's state by q. the r execution contains a last call to the *acknowledged_reading* primitive to read $\mathcal{R}_{pq}$. We name t_r the ending time of this call; at time t_r, we have $Local\_Reg_{qp}.copy\_state(q) \neq st.1.p$; thus $t_r < et(1,p)$. Between t_r and $et(1,p)$, a complete execution of the *acknowledged_reading* primitive to read $\mathcal{R}_{pq}$ has be done in order to obtain $Local\_Reg_{qp}.copy\_state(q) = st.1.p$ at time $et(1,p)$. Therefore, r finishes before the time $et(1,p)$. □

Theorem 6.7 *Let q and r be two neighbours of processor p. Let t_q (resp. t_r) be a time where a call of τ(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 $t_r > mt(i+1,p)$ then $i' > i \geq 1$ otherwise $i' = i \geq 1$ (see Lemma 6.5 and lemma 6.6). □

Lemma 6.8 *Only the first* τ(ATOMIC-STATE-READ) *of* p*'s state by* q *can be a* WRONG-READ.

Proof: Assuming that the WRONG-READ is not the first call of τ(ATOMIC-STATE-READ) by q of p's state. The preceding τ(ATOMIC-STATE-READ) contains a last call to the *acknowledged_reading* primitive to read $\mathcal{R}_{pq}$. We name t the ending time of this call. Between time t and $mt(1,p)$, we have $Local\_Reg_{qp}.copy\_state(q) \neq st.1.p$ or $\mathcal{R}_{pq}.copy\_flag = 0$. At the stating time of a WRONG-READ, we should have $\mathcal{R}_{qp}.copy\_state = st.1.p$, and $\mathcal{R}_{pq}.local\_flag = 1$ There is a contradiction. □

Between the time $mt(1,p)$ and the time $et(1,p)$, a call of τ(ATOMIC-STATE-READ) to get the p's state can return $st.1.p$, and another call (by another p's neighbour named q) that terminate after the first one can return another value (i.e. the initial value of $Local\_Reg_{pq}.local\_state(q)$, the initial value of $\mathcal{R}_{pq}$.local_state, the initial value of $Local\_Reg_{pq}.local\_state(p)$, 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 $\mathcal{A}lg$ in $\mathcal{A}$. The compiled algorithm in atomic-link model needs only bounded link registers if $\mathcal{A}lg$ requires only bounded state registers. An ATOMIC-STATE-WRITE operation requires at least $4 \times |\mathcal{N}.p|$ ATOMIC-LINK-READ and ATOMIC-LINK-WRITE operations. An ATOMIC-STATE-READ operation requires at least $|\mathcal{N}.p|$ ATOMIC-LINK-READ and ATOMIC-LINK-WRITE operations. But there is not limitation on the number of operations performed during an ATOMIC-STATE-READ or during an ATOMIC-STATE-WRITE operation. The duration of the τ(ATOMIC-STATE-WRITE) on p depends on the speed of p's neighbour (more precisely, on how often, they read the p's registers). The τ(ATOMIC-STATE-READ) also takes time, a processor may be locked for sometimes, before obtaining a neighbour state.

7 Other Compilers for basic Networks

Section 6 presented a self-stabilizing (but not wait-free) compiler from atomic-state system to atomic-link system. Denote this compiler the AS-AL compiler. Section 5 showed that there is no wait-free compiler from atomic-state to atomic-link. In this section we investigate the existance of compilers, and their fault-tolerance, between the other basic netowrk models.

Compiling from atomic-link networks to regular-link networks Lamport [18] presented a wait-free implementation of an atomic single-writer/single-reader register with regular single-writer/single-reader registers. This transformer requires two regular registers; one written by the writer and the other written by the reader. The relationship between the atomic-link model and the regular-link model is an instance of the relationship between atomic single-writer/single-reader register and regular single-writer/single-reader register. Thus, Lamport's implementation constitutes a wait-free compiler, calledd AL-RL, from atomic-link networks to regular-link networks. It is straightforward to confirm that AL-RL is also self-stabilizing.

Compiling from regular-state networks to regular-link networks: A natural way to transform an algorithm for a state model into an algorithm for a link model is the transformation:

τ(STATE-WRITE($\mathcal{R}_p$, ν)) $\equiv$ for every q in neighbourhood of p,
LINK-WRITE($\mathcal{R}$pq, ν)
τ(STATE-READ($\mathcal{R}_q$)) $\equiv$ $\nu \longleftarrow$LINK-READ($\mathcal{R}_{qp}$) ; return ν
The Naive Transformation

Attiya and Welch ([27] page 222 figure 10.2) show that the Naive Transformation is not a compiler from atomic-state systems to atomic-link systems. Notice, however, that under the Naive Transformation, the implementation of any READ that overlaps the implementation of a WRITE will return either the value being written or the one most recently written before the READ began. Furthermore, this holds even if the target system has only regular registers instead of atomic registers on its links. Thus, the returned value is exactly what is allowed by the definition of a regular register. Finally, notice that The Naive Transformation is a wait-free transformation. It is also easily confirmed to be self-stabilizing. So we conclude:

Observation 7.1 *The Naive Transformation is a wait-free and self-stabilizing compiler from the regular-state system to the regular-link It is not a compiler from the atomic-state system to the atomic-link (and hence not to the regular-link) system.*

Therefore, if an algorithm for the atomic-state model works correctly unchanged on the regular-state model, then the Naive transformation of the algorithm will be correct for the regular-link model. The Naive transformation is called the RS-RL compiler when applied to regular-state networks.

Compiling from regular-state networks to atomic-state networks The following identity transformation implements a regular-state system on an atomic-state system.

τ(REGULAR-STATE-WRITE($\mathcal{R}_p$, ν)) $\equiv$ ATOMIC-STATE-WRITE($\mathcal{R}_p$, ν)
τ(REGULAR-STATE-READ($\mathcal{R}_q$)) $\equiv$ $\nu \longleftarrow$ATOMIC-STATE-READ($\mathcal{R}_{qp}$) ; return ν
Compiler RS-RL

Compiler RS-RL is both wait-free and self-stabilizing.

Compiling from regular-link networks to atomic-link networks The identity transformation also provides a wait-free, self-stabilizing compiler from a regular-link system to an atomic-link system, denoted the RL-AL.

Compiling from atomic-link networks to atomic-state networks To implement an atomic-link system on an atomic-state network can be easily done provided the processors in the network have distinct label within distance two.

structure of $\mathcal{R}_p$ = array of Δ elements. $\mathcal{R}_p$[i] has two fields: *name* field and the *value* field.

τ(ATOMIC-LINK-WRITE($\mathcal{R}_{pq}$, ν)) $\equiv$ $AR \longleftarrow$ATOMIC-STATE-READ($\mathcal{R}_p$);
i := 0; **while** (AR[i].name $\neq id_q$) **do** i++; **done**
AR[i].value := ν; ATOMIC-STATE-WRITE($\mathcal{R}_p$, AR)

$\tau(\text{ATOMIC-LINK-READ}(\mathcal{R}_{qp})) \equiv AR \longleftarrow \text{ATOMIC-STATE-READ}(\mathcal{R}_{q})$;
i := 0; **while** $(AR[\text{i}].\text{name} \neq id_q)$ **do** i++; **done**
return $AR[i].value$

Compiler AL-AS

This compiler, called AL-AS, is wait-free and self-stabilizing.

Compiling from regular-link networks to regular-state networks The compiler, called RL-RS, is very similar to the previous one. It is wait-free and self-stabilizing, and also requires the processors of the network have distinct label within distance two.

Compiling from atomic-state networks to regular-state networks The implementation of an atomic-state system on the corresponding regular-state network can be done by composing combining a AS-AL, a AL-RL and a RL-RS compiler. This compiler is self-stabilizing but not wait-free. There is no wait-free compiler from an atomic-state system into the corresponding regular-state network, since otherwise, the composition of it with RS-RL, and RL-AL, would be a wait-free compiler from atomic-state to atomic-link, in contradiction with Theorem 5.2.

All these transformation are summarized in the figure 7.

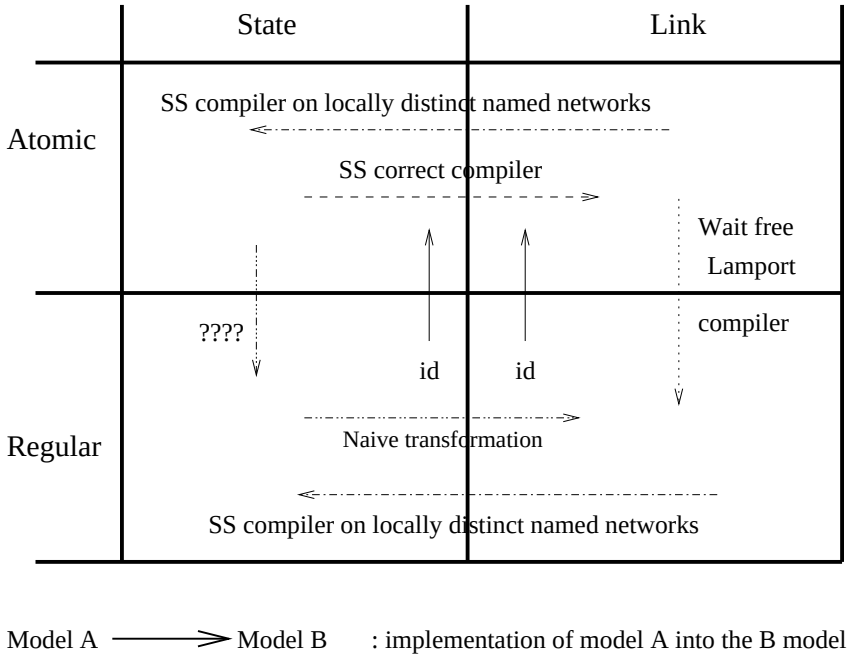

Figure 1: Transformation from one network model to other ones

References

[1] G Antonoiu and PK Srimani. Mutual exclusion between neighboring nodes in an arbitrary system graph tree that stabilizes using read/write atomicity. In *Euro-Par'99 Parallel Processing, Proceedings LNCS:1685*, pages 823–830, 1999.

[2] C Boulinier, F Petit, and V Villain. When graph theory helps self-stabilization. In *PODC04 Proceedings of the twenty-third Annual ACM Symposium on Principles of Distributed Computing*, pages 150–159. ACM Press, 2004.

[3] S Chaudhuri, M Kosa, and J Welch. Upper and lower bounds for one-write multivalued regular registers. In *Proceedings of the 3rd IEEE Symposium on Parallel and Distributed Processing*, pages 134–141, 1991.

[4] EW Dijkstra. Self stabilizing systems in spite of distributed control. *Communications of the Association of the Computing Machinery*, 17(11):643–644, 1974.

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

[6] 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.

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

[8] S Dolev and JL Welch. Wait-free clock synchronization. In *PODC93 Proceedings of the Twelfth Annual ACM Symposium on Principles of Distributed Computing*, pages 97–107. ACM, 1993.

[9] MG Gouda and F Haddix. The linear alternator. In *Proceedings of the Third Workshop on Self-Stabilizing Systems*, pages 31–47. Carleton University Press, 1997.

[10] MG Gouda and F Haddix. The alternator. In *Proceedings of the Fourth Workshop on Self-Stabilizing Systems (published in association with ICDCS99 The 19th IEEE International Conference on Distributed Computing Systems)*, pages 48–53. IEEE Computer Society, 1999.

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

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

[13] 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.

[14] ST Huang. The fuzzy philosophers. In *Parallel and Distributed Processing (IPDPS Workshops 2000), Springer LNCS:1800*, pages 130–136, 2000.

[15] A Israeli and M Li. Bounded time-stamps. *Distributed Computing*, 6(4):205–209, 1993.

[16] C Johnen, I Lavallée, and C Lavault. Fair and reliable self-stabilizing communication. In *4th International Conference On Principles Of DIstributed Systems, OPODIS'2000*, pages 163–176. Studia Informatica Universalis, 2000.

[17] H Kakugawa and M Yamashita. Self-stabilizing local mutual exclusion on networks on which process identifiers are not distinct. In *SRDS 2002 21st Symposium on Reliable Distributed Systems, IEEE Computer Society Press*, pages 202–211, 2002.

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

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

[20] M Mizuno and M Nesterenko. A transformation of self-stabilizing serial model programs for asynchronous parallel computing environments. *Information Processing Letters*, 66(6):285–290, 1998.

[21] S Moriya, M Inoue, T Masuzawa, and H Fujiwara. Self-stabilizing wait-free clock synchronization with bounded space. In *2nd International Conference On Principles Of DIstributed Systems, OPODIS'98*, pages 129–143, 1998.

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

[23] M Nesterenko and A Arora. Stabilizing dining philosophers that tolerate malicious crashes. In *ICDCS02 The 22nd IEEE International Conference on Distributed Computing Systems*, pages 191–!98, 2002.

[24] M Papatriantafilou and P Tsigas. On self-stabilizing wait-free clock synchronization. *Parallel Processing Letters*, 7(3):321–328, 1997.

[25] AK Singh, JH Anderson, and MG Gouda. The elusive atomic register. *Journal of the Association of the Computing Machinery*, 41(2):311–339, 1994.

[26] PMB Vitanyi. Simple wait-free multireader registers. In *DISC02 Distributed Computing 16th International Symposium, Springer LNCS:2508*, pages 118–132. Springer-Verlag, 2002.

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