Fault-tolerant Implementations of regular Registers by safe Registers in Link Model

Colette Johnen
LRI, Univ. Paris-Sud, CNRS, F-91405 Orsay, France - colette@lri.fr

Lisa Higham
Computer Science Department, University of Calgary, Canada - higham@cpsc.ugalgary.ca

February 12, 2008

Abstract

A network that uses locally shared registers can be modelled by a graph where nodes represent processors and there is an edge between two nodes if and only if the corresponding processors communicate directly by reading or writing registers shared between them. Two variants are defined by A variant of the model assumes that registers are single-writer/single-reader and are located on the edges (called link models).

This paper is concerned with the three link network models that arise from specifying the type of shared registers (safe, regular, or atomic). Specifically, we seek to determine under what conditions and with what fault-tolerance guarantees it is possible to transform a solution under one of these models into a solution under models.

The fault tolerant properties we consider are self-stabilization and wait-freedom. Our principal result is a wait-free and self-stabilizing compiler from the regular-link model to the safe-link model.

For all the remaining relationships among these three models under either self-stabilizing and wait-free requirements, we either observe that they have been answered by existing research. Thus, our compiler closes the proposed questions among the three models. For instance, any self-stabilizing algorithms designed for the atomic-link model (also called R/W atomicity model) can be implemented using safe registers instead of atomic registers.

Keywords: network models, distributed algorithms, safe registers, regular registers, atomic registers link-register models, self-stabilization, wait-freedom.

1 Introduction

This paper address this question for networks of processors that communication by locally shared registers. A network that uses locally shared registers can be modelled by a graph where nodes represent processors and there is an edge between two nodes if and only if the corresponding processors communicate directly by reading or writing registers shared between them. Two variants are defined by specifying whether the registers are multi-reader and located at the nodes (called state models) or single-reader and located on the edges (called link models).

The shared registers used by the communicating processors further distinguishes possible models. Lamport [10] defined three models of registers, differentiated by the possible outcome of read operations that overlap concurrent write operations. These three register types, in order of increasing power, are called safe, regular, and atomic. Program design is easier assuming atomic registers rather than regular registers but the hardware implementation of an atomic register is costlier than the implementation of a regular register. Safe registers are cheaper still; they capture a notion of directly sensing the hardware.

By specifying either state or link communication, via shared registers that are either safe, regular, or atomic, we arrive at six different network models

that use locally shared registers. For example, the regular-link model has regular registers located on the edges of the network. The other models are named similarly.

An algorithm for any one of these networks could provide some fault tolerance. So, we consider a third parameter, namely, wait-freedom, which captures tolerance of stopping failures of components of the network, or self-stabilization, which captures recovery of the network from transient errors of its components.

Related research Due to one of Lamport's seminal papers [10] and several other subsequent papers [1, 6, 11, 2], it is already known how to construct wait-free, multi-writer multi-reader, shared atomic registers from only a collection of safe bits each shared between a single-writer and a single-reader. However, these constructions are not self-stabilizing. Hoepman, Papatrianfafiou and Tsigas [8] presented self-stabilizing versions of some of these well-known implementations. For instance, they present a wait-free and self-stabilizing implementation of a single-writer/single-reader regular binary register using a single-writer/dual-reader safe binary register. In [8], it was established that the following impossibility result : there is not wait-free and self-stabilizing implementation of single-writer/single-reader regular binary register by a single-writer/single-reader safe binary register.

In previous works, we have established that there is no general wait-free compiler from atomic-state networks to atomic-link networks in [7], and no general wait-free compiler from atomic-state networks to regular-link networks in [9]. The proofs proceed by showing that any such compiler would require shared registers between any two processors, which is not the case in general networks. 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. In [9], a self-stabilizing compiler from the atomic-state model to the regular-state model is presented. This compiler is also *silent* [4]. That is, if, once registers are stabilized, the atomic-state algorithm does not require the participation of neighbours, then the transformed regular-state algorithm also does not require the participation of neighbours. As a consequence, our compiler does not add significant overhead to communication.

Contributions of this paper Our principal result is a wait-free and self-stabilizing compiler from the regular-link model to the safe-link model.

Paper overview Section 2 defines the six basic models we are considering, contains several definitions required for the rest of the paper, and presents a formal definition of a compiler from one register-based model to another. In Section 3, we present a wait-free and self-stabilizing compiler from 1W/1R pseudo-regular-link register to 1W/1R safe-link registers. In Section 4, we present a wait-free and self-stabilizing compiler from 1W/1R regular-link register to 1W/1R pseudo-regular-link registers. The combination of two compilers (presented in Section 5) provides a wait-free and self-stabilizing compiler from distributed networks where neighbours communicate via regular-link registers to distributed networks where neighbours communicate via safe-link registers.

2 Definitions and Models

2.1 Distributed Systems

Shared registers. Let R be a single-writer/multi-reader register that can contain any value in domain T. R supports only the operations READ and WRITE. Each READ and WRITE operation, o, has a time interval corresponding to the time between the invocation of o, denoted inv(o), and the response of o, denoted resp(o). An operation o *happens-before* operation o' READ operations,may overlap a WRITE. Lamport [10] defined several kinds of such registers depending on the semantics when READ and WRITE operations overlap. Register X is *safe* if each READ that does not overlap any WRITE returns the value of the latest WRITE that happens-before it, and otherwise returns any value in T. Register R is *regular* if it is safe and any READ that overlaps a WRITE returns the value of either the latest WRITE that happens-before it, or the value of some overlapping WRITE.

Register R is *atomic* if it is regular, and if any READ, r, overlaps a WRITE, w, and returns the value written by w, then any READ, r', that happens-after r

must not return the value of any WRITE that happens-before w.

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 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 *state* network models, each processor p owns a single-writer multi-reader shared register R_p, which is writable by p and readable by each of p's neighbours.

In the *link* network models, for each edge $\langle pq \rangle \in E$, there are two single-writer single-reader registers. Register R_{pq} is writable by p and readable by q; register R_{qp} is writable by q and readable by p.

Each of these state and link model types is further refined by specifying whether the shared registers are atomic, regular, or safe. Thus, there are six different network models that arise by specifying two parameter for the shared registers: Strength $\in$ {atomic, regular, safe} and Location $\in$ {link, state}. We name the six register-based models as in indicated in Table 1.

	state models	*link models*
atomic registers	atomic-state	atomic-link
regular registers	regular-state	regular-link
safe registers	safe-state	safe-link

Table 1: Six register-based network models

Given a graph G, we use Strength-Location(G) to denote the network with topology G and network model Strength-Location. For example Regular-Link(G) is the network with topology G that has regular single-writer mono-reader shared registers located at each eadge. We use similar notation for the WRITE and READ operations on each of these models. For example, in an regular-link network model, the WRITE and READ operations are denoted:

- RL-WRITE(R, ν) to denote the write of value ν to the shared register R.
- $\nu \longleftarrow$RL-READ(R) to denote the read of the shared register R that returns the value ν.

The atomic-link model is identical to a model used by Dolev, Israeli and Moran [5]. The atomic-state model has been assumed by several others in subsequent papers [12].

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*. We use the term *network* to mean just the topology and the communication model and *system* to mean the network together with the algorithm. Of course, the assigned program must use only the 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.

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 Alg *solves problem* P *on network class* $\mathcal{N}$ if for any network $N \in \mathcal{N}$ all computations of algorithm Alg on N satisfies predicate P.

2.2 Fault-tolerance

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. 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.

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 configurations. Let P be a predicate defined on configurations. The set of configurations satisfying P is an *attractor* if and only if

- **convergence:** starting from any configuration, any computation reaches a configuration satisfying P.
- **closure:** For any configuration C satisfying P, the successor configuration reached by any computation step applied to C also satisfies P.

Let PS be a predicate defined on computations. A distributed system is *self-stabilizing to* PS if and only if there is a predicate, Leg, on configurations such that:

- **convergence and closure:** The set of configurations satisfying Leg is an attractor.
- **correctness:** Any computation from a configuration satisfying Leg satisfies PS.

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

2.3 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. To transform an algorithm for Regular-Link(G) to an algorithm for Safe-Link(G) we replace each RL-WRITE and RL-READ by every processor p with a program for p that uses only local operations and the operations SL-WRITE and SL-READ. Thus a *program transformation* from Regular-Link(G) to Safe-Link(G) is just a mapping τ where τ(RL-WRITE(R, ν)) and τ(RL-READ(R)) are programs whose operations are on registers in Safe-Link(G) and such that τ(RL-READ(R)) returns a value.

We are concerned with program transformations that preserve correctness. Since correctness is defined by a predicate on computations, and the computations differ in each network model, we need to make precise what is meant by "preserves correctness".

Consider a specified system S. More formally, let $S = (G, \mathcal{N}, P)$ where G is a graph and P is a collection of programs, one for each node of G. We can associate a set of computations, C, with S in the natural way: C is just the set of all computations that can arise by executing the programs in P on the network $N = \mathcal{N}(G)$.

Now let τ be a transformation from $\mathcal{N}$ to $\widehat{\mathcal{N}}$. Given τ, there is another way to associate a set of computations with S. Denote by $\tau(P)$ the set of all the programs in P after being transformed by τ. Any computation of the target system $T = (G, \widehat{\mathcal{N}}, \tau(P))$ can be *interpreted* as a computation of S by attaching the value returned by each τ(READ) to the corresponding READ invocation. (Such a computation looks just like a computation of S except the value returned by each READ is obtained via the transformation τ instead of directly by executing S.)

For correctness of τ we require that this derived computation is allowed by S. In that case, we say that τ is an *implementation* of S on T.

Let A denote a collection of algorithms for network model $\mathcal{N}$. A transformation τ is a *compiler for* A *from* $\mathcal{N}$ *to* $\widehat{\mathcal{N}}$ if τ is an implementation of $S = (G, \mathcal{N}, P)$ on $T = (G, \widehat{\mathcal{N}}, \tau(P))$ for any $P \in A$ and any graph G. A transformation is a *self-stabilizing compiler* (resp. *wait free compiler* from $\mathcal{N}$ to $\widehat{\mathcal{N}}$ if it is a compiler from $\mathcal{N}$ to $\widehat{\mathcal{N}}$ and, it maps self-stabilizing systems to self-stabilizing systems (resp. it maps wait-free systems to wait-free systems).

2.3.1 How to prove the correctness of self-stabilizing compiler

There are two major components of the proof of self-stabilizing compiler: termination and correctness.

Termination: In the self-stabilizing framework it is possible that initially the program counters of some processors are inside their τ(RL-WRITE) or τ(RL-READ) programs and their register values are corrupted and inconsistent. In this case, some

τ(RL-WRITE) and τ(RL-READ) programs are only partially executed. So it is essential to establish that any complete or partial execution of τ(RL-WRITE) and τ(RL-READ) terminates.

Correctness: Consider a specified system $S = (G, \mathcal{N}, Alg)$ where $Alg \in A$ and the target system $T = (G, \widehat{\mathcal{N}}, \tau(Alg))$ that S is transformed to by the transformation τ. We would like to show that the possible computations of T correspond to computation of S, or, more precisely, that the interpretation of any computation of T is a computation of S. Actually, we cannot quite achieve this goal because the algorithms being considered are self-stabilizing. So we show correctness of τ in two substeps. First we show (in **Convergence**) that the set of legitimate configurations is an attractor. Next we show that, starting from any legitimate configuration, any computation from that point on has an interpretation as a computation of S.

3 Compiler from pseudo-regular-link to safe-link

Register R is *pseudo-regular* if it is safe and any READ that overlaps a **single** WRITE returns the value of either the latest WRITE that happens-before it, or the value of the overlapping WRITE.

Let A be the set of algorithms for the pseudo-regular-link model that satisfy: every processor p, for any p's neighbour, named q, executes $\tau 1$(PSEUDO-RL-WRITE(R_{pq}, -)) at least once after any transient failure.

We will show that Algorithm 1 is a wait-free and self-stabilizing compiler from pseudo-regular-link networks to safe-link networks for all algorithms in A.

During the execution of $\tau 1$(PSEUDO-RL-WRITE(R_{pq}, -)) the written value is written into three distinct safe registers (named $R1_{pq}$, $R2_{pq}$, and $R3_{pq}$). During the execution of $\tau 1$(RL-READ(R_{pq})) the same three safe registers are readed in the opposiiter order.

Let us name PR-read, an execution of $\tau 1$(RL-READ(R_{pq})). Let us name PR-write, the latest execution of $\tau 1$(PSEUDO-RL-WRITE(R_{pq}, v)). If it exists, let us name PR-write', the single execution of $\tau 1$(PSEUDO-RL-WRITE(R_{pq}, v')) that overlaps PR-read. PR-read has to return a suitable value (meaning v or v'). The safe registers are accessed in the opposite order by PR-write and PR-read; thus at most a single safe register of $\{R1_{pq}, R2_{pq}, R3_{pq}\}$ can be read by PR-read when there is an overlapping write to the same register by PR-write'.

Compiler 1 Code of Self-stabilizing compiler from pseudo-regular-link networks to safe-link networks

The 1W, 1R pseudo-regular register (R_{pq}) is replaced by 3 1W, 1R safe registers: $R1_{pq}$, $R2_{pq}$, and $R3_{pq}$.

Code on the processor p:

$\tau 1$(PSEUDO-RL-WRITE(R_{pq}, new_s))
  [** begin of pre section **]
   SL-WRITE($R1_{pq}$, new_s);
  [** end of pre section, **]
  [** begin of unsafe section **]
   SL-WRITE($R2_{pq}$, new_s);
  [** end of unsafe section, **]
  [** begin of post section **]
   SL-WRITE($R3_{pq}$, new_s);
  [** end of post section **]

$\tau 1$(RL-READ(R_{qp}))
  $v1$, $v2$, and $v3$ are local variables of the function.
   $v3 \leftarrow-$SL-READ($R3_{qp}$);
   $v2 \leftarrow-$SL-READ($R2_{qp}$);
   $v1 \leftarrow-$SL-READ($R1_{qp}$);
   if ($v3$==$v2$) or (v1==v2) then return $v2$
   else return $v1$; **fi**

If the $R3_{pq}$ read is overlapped then the subsequent read of $R2_{pq}$ and of $R1_{pq}$ will be the value just written by PR-write' ($v1 = v2 = v'$). If the $R2_{pq}$ read is overlapped then the subsequent read of $R1_{pq}$ will be the value just written by PR-write' ($v1 = v'$); and the previous read of of $R3_{pq}$ will be the value written by PR-write ($v3 = v$). At the time of the read of $R3_{pq}$, PR-write did not start to write in this safe registers. If the $R1_{pq}$ read is overlapped then the previous reads of of $R3_{pq}$ and $R2_{pq}$ will be the value written by PR-write ($v3 = v2 = v$). At the time of the reads, PR-write did not start to write in the safe

registers of $R3_{pq}$ and $R2_{pq}$. Therefore, it is possible to ensure that PR-read returns v or v'. Hence the value return by PR-read satisfies the requirement of a pseudo-regular register.

3.1 Proof of Compiler 1

Let p and q be two neighbour processors. In this section, all registers are 1W and 1R; the writer is processor p and the reader is q. Also, the register REG_{pq} is simplely denoted REG.

3.1.1 Termination

In this section, we prove that any execution (partial or complete) of $\tau 1$(PSEUDO-RL-WRITE(R,-)) and $\tau 1$(PSEUDO-RL-READ(R)) terminates.

Lemma 3.1 *Any $\tau 1$(PSEUDO-RL-WRITE(R,-)) execution by p terminates.*

Proof: During the execution of $\tau 1$(PSEUDO-RL-WRITE(R,-)), p performs at most three SL-WRITE operations. □

Lemma 3.2 *Any $\tau 1$(PSEUDO-RL-READ(R)) execution by p terminates.*

Proof: During the execution of $\tau 1$(PSEUDO-RL-READ(R)), p performs at most three SL-READ operations and a internal operation. □

Theorem 3.1 *If $\tau 1$ is a compiler from pseudo-regular-link model to safe-link model then $\tau 1$ is a wait-free compiler.*

Proof: Any $\tau 1$(PSEUDO-RL-READ(R)) or $\tau 1$(PSEUDO-RL-WRITE(R,-)) is done in finite number of steps regardless of other processor actions. □

3.1.2 Legitimate Configurations

In this section, we will prove the set of configurations verifying $Leg1$ is an attractor.

Definition 3.1 $L1_s(p) \equiv$ *[$R3 == R2 \wedge$ p's program_counter is in the pre section of $\tau 1$(PSEUDO-RL-WRITE(R,-))]*
$L2_s(p,q) \equiv$ *[$R1 == new\_state \wedge$ p's program_counter is in the unsafe section of $\tau 1$(PSEUDO-RL-WRITE(R,-))]*
$L3_s(p,q) \equiv$ *[$R1 == R2 == new\_state \wedge$ p's program_counter is in the post section of $\tau 1$(PSEUDO-RL-WRITE(R,-))]*
$L4_s(p,q) \equiv$ *[$R1 == R2 == R3 == new\_state \wedge$ p's program_counter is not in code of $\tau 1$(PSEUDO-RL-WRITE(R,-))]*

$Correct\_state1(p,q) \equiv L1_s(p,q) \vee L2_s(p,q) \vee L3_s(p,q) \vee L4_s(p)$

$Leg1 \equiv (\forall (p,q) \in E\ Correct\_state1(p,q) \equiv True)$.

Lemma 3.3 *$Correct\_state1(p,q)$ is closed*

Proof: $L4_s(p,q)$ stays verified till p is not starting a $\tau 1$(PSEUDO-RL-WRITE(R,-)) execution, because the value of $R1$, $R2$, and $R3$ are not modified. If $L4_s(p,q)$ is verified then we have where p enters in the pre section. Thus $L1_s(p,q)$ is verified.

During the pre section, only the value of $R1$ is modified; thus, $L1_s(p,q)$ is verified till p's counter stays in the pre section if $L4_s(p,q)$ was verified before entering in the pre section. When p's program counter exits of the pre section, we have $R1 == new\_state$ and the p's program counter is in the unsafe section. Thus $L2_s(p,q)$ is verified.

$L2_s(p,q)$ stays verified till p's counter stays in the unsafe section, because the value of $R1$ is not modified during the unsafe section. When p's program counter exits of the unsafe section, we have $R2 == new\_state$ and the p's program counter is in the post section. Thus $L3_s(p,q)$ is verified, if $L2_s(p,q)$ was verified when p's program counter was in the unsafe section.

$L3_s(p,q)$ stays verified till p's counter stays in the post section, because only the value of $R3$ is modified during the post section. When p's program counter exits of the post section, we have $R3 == new\_state$. Thus $L4_s(p,q)$ is verified, if $L3_s(p,q)$ was verified when p's program counter was in the post section. □

Lemma 3.4 *Let A be the set of algorithms for the pseudo-regular-link model that satisfy: every processor p, for any p's neighbour, named q, executes $\tau 1$(*PSEUDO-RL-WRITE(R,-)*) at least once after any transient failure. Let $Prot$ be a protocol of A. The set of configuration verifying $Leg1$ is an attractor of target system $T = (G,$ safe-link$, \tau 1(Prot))$*

Proof: We need to prove that any execution of T reaches a configuration where $Correct\_state1(p,q)$ is verified.

Let us study the first complete execution of $\tau 1$(PSEUDO-RL-WRITE(R,-)) done after a transient failure. Such an execution exists because $Prot$ belongs to A.

When p's program counter exits of the pre section, we have $R1 == new\_state$ and the p's program counter is in the unsafe section : thus $L2_s(p,q)$ is verified. □

3.1.3 Correctness

Consider a specified system $S = (G,$ pseudo-regular-link$, Alg)$ where $Alg \in A$. S is transformed by the transformation $\tau 1$ (i.e. Compiler 1) to $T = (G,$ safe-link$, \tau 1(Alg))$

In this section, we will establish that any computation of T from a legitimate configuration, has an interpretation as a computation of S.

Definition 3.2

- $st1(i)$ *denote the start time of the ith call of* $\tau 1$*(*PSEUDO-RL-WRITE(R,-)*). If the ith call of* $\tau 1$*(*PSEUDO-RL-WRITE(R,-)*) does not exist then* $st1(i)$ *has the value* $+\infty$.
- $et1(i)$ *denotes the end time of the ith call of* $\tau 1$*(*PSEUDO-RL-WRITE(R,-)*) by processor p. If the ith call of* $\tau 1$*(*PSEUDO-RL-WRITE(R,-)*) does not exist then* $st1(i)$ *has the value* $+\infty$.
- *The written* $value$ *during the ith execution of* $\tau 1$*(*PSEUDO-RL-WRITE(R,-)*) is denoted PR-value*(i).

Observation 3.1 *For $i > 0$, at time $et1(i)$, $L4_s(p,q)$ is verified and value of $R2$ is PR-value(i).*

*Before $st1(1)$, the register $R2$ may have two distinct values : its initial value and the written value during the single partial execution of $\tau 1$(*PSEUDO-RL-WRITE(R,-)*).*

Definition 3.3 *We denote by PR-value(-1) the initial value of $R2$. We denote by $et1(-1)$ the time 0.*

*If there exists a partial execution of $\tau 1$(*PSEUDO-RL-WRITE(R,-)*) then $et1(0)$ denotes the end time of this partial execution, and we define (1) $st1(0)$ has the time 0. If the partial execution writes a value in $R2$ then we denoted by PR-value(0) the written value, otherwise PR-value(0) the initial value of $R2$.*

*If there does not exist a partial execution of $\tau 1$(*PSEUDO-RL-WRITE(R,-)*) then $et1(0)$ is defined has the time 0. We denoted by PR-value(0) the initial value of $R2$.*

Observation 3.2 *Once $Correct\_state1(p,q)$ is verified, at any time of the interval $[et1(i), et1(i+1)]$ the value of $R1$ (resp. $R2$, and $R3$) is PR-value(i) or PR-value$(i+1)$.*

Then correctness is achieved if (1) any $\tau 1$(PSEUDO-RL-READ(R)) that is not overlapped by a $\tau 1$(PSEUDO-RL-WRITE(R,-)) returns the written value by the latest $\tau 1$(PSEUDO-RL-WRITE(R,-)) that happens-before it; and if (2) a $\tau 1$(PSEUDO-RL-READ(R)) overlapped by a single $\tau 1$(PSEUDO-RL-WRITE(R,-)) returns the written value of either the latest $\tau 1$(PSEUDO-RL-WRITE(R,-)) that happens-before it, or of the overlapping $\tau 1$(PSEUDO-RL-WRITE(R,-)). More precisely, The following properties have to be proven to establish the correctness. Once that $Correct\_state1(p,q)$ is verified, for any $i \geq -1$,

- any $\tau 1$(PSEUDO-RL-READ(R)) starting after $et1(i)$ and terminating before $st1(i+1)$ retuns PR-value(i).
- any $\tau 1$(PSEUDO-RL-READ(R)) starting after $et1(i)$ and terminating before $st1(i+2)$ retuns PR-value(i) or PR-value($i+1$).

Lemma 3.5 *Once that $Correct\_state1(p,q)$ is verified, for any $i \geq -1$, any $\tau 1$(PSEUDO-RL-READ(R)) starting after $et1(i)$ and terminating before $st1(i+1)$ retuns PR-value(i).*

Proof: During the execution of $\tau 1$(PSEUDO-RL-READ(R)), $L4_s(p,q)$ is always verified, and no value is written in the safe register $R2$. Thus, the return value is the value of $R2$ at time $et1(i)$. At time $et1(i)$, the value of $R2$ is PR-value(i) (by definition of PR-value($i+1$)). □

Lemma 3.6 *Once that $Correct\_state1(p,q)$ is verified, for any $i \geq -1$, any $\tau 1$(PSEUDO-RL-READ(R)) starting after $et1(i)$ and terminating before $st1(i+2)$ retuns PR-value(i) or PR-value($i+1$).*

Proof: We will do a proof by contradiction.

Let us name PR-read, a $\tau 1$(PSEUDO-RL-READ(R)) execution starting after $et1(i)$ and terminating before $st1(i+2)$ that does not return PR-value(i) either PR-value($i+1$).

Case 1: PR-read returns $v1$. According to $\tau 1$(PSEUDO-RL-READ(R)) code, at the end of PR-read, we have $v2 \neq v3$.

$v1$ is not equal to PR-value(i) value and is not equal to PR-value($i+1$) value only if the reading of $R1$ overlaps the execution of pre section done during the $i+1$th call of $\tau 1$(PSEUDO-RL-READ(R)) if $i \geq 0$ or otherwise during the partial execution of $\tau 1$(PSEUDO-RL-READ(R)). At the starting time of the reading of $R1$ done during PR-read, the execution of unsafe section has not started. The reading of $R3$ and of $R2$ precedes the reading of $R1$, thus at the ending time of $R2$ reading, the execution of unsafe section has not started. We conclude that during the reading of $R3$ and $R2$, none writing is done in these registers and the predicate $L1_s(p,q)$ or $L4_s(p,q)$ is verified. At the end of PR-read, we have $v2 == v3$. There is a contradiction.

Case 2: PR-read returns $v2$. According to $\tau 1$(PSEUDO-RL-READ(R)) code, at the end of PR-read, we have $v2 == v3$ or $v2 == v1$. $v2$ is not equal to PR-value(i) value and is not equal to PR-value($i+1$). value only if the reading of the safe register $R2$ overlaps the execution of unsafe section done during the $i+1$th call of $\tau 1$(PSEUDO-RL-READ(R)) if $i \geq 0$ or otherwise during the partial execution of $\tau 1$(PSEUDO-RL-READ(R)).

At the starting time of the reading of $R2$, the execution of post section has not started. The reading of$R3$ precedes the reading of $R2$, thus at the ending time of $R3$ reading, the execution of post section has not started. We conclude that the reading of $R3$ cannot overlap the execution of post section. At the end of PR-read, $v3 ==$ PR-value(i) or $v3 ==$ PR-value($i+1$). Therefore, we have $v2 \neq v3$.

At the ending time of the reading of the safe register $R2$ The execution of pre section is terminated, because the execution of pre section precedes the execution of the unsafe section. Thus at the starting time of $R1$ reading, the execution of pre section is terminated. The reading of $R1$ cannot overlap the execution of pre section. We conclude, that at the end of PR-read, $v1 ==$ PR-value(i) or $v1 ==$ PR-value($i+1$). Therefore, we have $v2 \neq v1$.

At the end of PR-read, we have $v2 \neq v1$ and $v2 \neq v3$. There is a contradiction. □

4 Compiler from regular-link to pseudo-regular-link

Let A be the set of algorithms for the regular-link model that satisfy: every processor p, for any p's neighbour, named q, executes $\tau 2$(RL-WRITE(R_{pq}, -)) at least once after any transient failure.

We will show that Algorithm 2 is a wait-free and self-stabilizing compiler from regular-link networks to pseudo-regular-link networks for all algorithms in

Compiler 2 Code of Self-stabilizing compiler from regular-link networks to pseudo-regular-link networks

$Flag[0..2]_{pq}$, $R[0..2]_{pq}$, and RC_{qp} are 1W, 1R pseudo-regular registers.

$\oplus$ is the addition modulo 3.

Code on the processor p:

$\tau 2$(RL-WRITE(R_{pq}, new_state))
$color$ is a local variable of the procedure.
    color $\longleftarrow$ PSEUDO-RL-READ(RC_{qp});
    writing(R_{pq}, new_state, color);

$\tau 2$(RL-READ(R_{qp}))
$f[0..2]$, $v[0..2]$, and c are local variables.
    for c := 0 to 2 **do**
        PSEUDO-RL-WRITE(RC_{pq}, c);
        (f[c],v[c]) $\longleftarrow$ reading(R_{qp}, c) ;
    done
    if ($f[0] == f[1] == 2$) **then** return($v[1]$) ;
    else return($v[2]$); **fi**

writing(R_{pq}, value, c) :
  PSEUDO-RL-WRITE($R[c]_{pq}$, value);
  [** begin of pre section **]
  PSEUDO-RL-WRITE($Flag[c \oplus 2]_{pq}$, c);
  [** end of pre section, begin of post section **]
  PSEUDO-RL-WRITE($Flag[c \oplus 1]_{pq}$, c);
  [** end of post section **]

reading(R_{qp}, c) :
f and v are local variables.
  f $\longleftarrow$ PSEUDO-RL-READ($Flag[c]_{qp}$);
  if $f \neq c \oplus 1$ **then** $f := c \oplus 2$; **fi**
  v $\longleftarrow$ PSEUDO-RL-READ($R[f]_{qp}$);
  return(f, v);

A.

If we could ensure that no more than one write could overlap a read operation, a pseudo-regular register would suffice in place of a regular register. For a single-reader single-writer model, this observation suggests that we try to avoid overlap by having several pseudo-registers available for the writer and arranging communication from the reader to direct the writer which one to use. To implement this idea in the pseudo-regular-link model, the regular-register R_{pq} is implemented with three pseudo-regular copies $R[i]_{pq}$ where $i \in \{0, 1, 2\}$. (only one of them contains the last written value.) Each link pq has a color values in $\{0, 1, 2\}$ written by the reader q and read by the writer p. Processor p implements an RL-WRITE to R_{pq} by writing to the copy $R[i]_{pq}$ if it believes the current color of the link is i. Three additional pseudo-regular registers are needed, $Flag[i]$ where $i \in \{0, 1, 2\}$, which are used to help the reader determine which of the three copies has the latest value.

The values of the pseudo-regulars $Flag[i]_{pq}$ where $i \in \{0, 1, 2\}$ help q to find out which of three registers $R[i]_{pq}$ contains the most recent value. The $Flag[i]_{pq}$ value "point" to the one having the most recent value of both registers $R[i \oplus 1]_{pq}$ and $R[i \oplus 2]_{pq}$.

In a $\tau 2$(RL-WRITE(R_{pq}, -)) execution, p first reads RC_{qp} to get a color $i \in \{0, 1, 2\}$. It then writes its new state to $R[i]_{pq}$, and set both registers $Flag[i \oplus 2]_{pq}$ and $Flag[i \oplus 1]_{pq}$ to i thus making them "point to" the register just written. Notice that during the execution of writing(R_{pq}, v, col), no write operation in a register of $Set(col)$ is done.

$\tau 2$(RL-READ(R_{pq})) is done in three steps. During the step i, only the registers of $Set(i)$ are read. The first action of the step i is to set the current color of the link to i (i.e. write operation in its output register RC_{pq}). Then, the step i is concluded by the execution of reading(R_{pq}, i). First action of reading(R_{pq}, i), is to find out which of both registers $R[i \oplus 2]_{pq}$ and $R[i \oplus 1]_{pq}$ has the more recent value. This piece of information is stored in the register $Flag[i]_{pq}$. Second action is to read the register having the more recent value between $R[i \oplus 2]_{pq}$ and $R[i \oplus 1]_{pq}$.

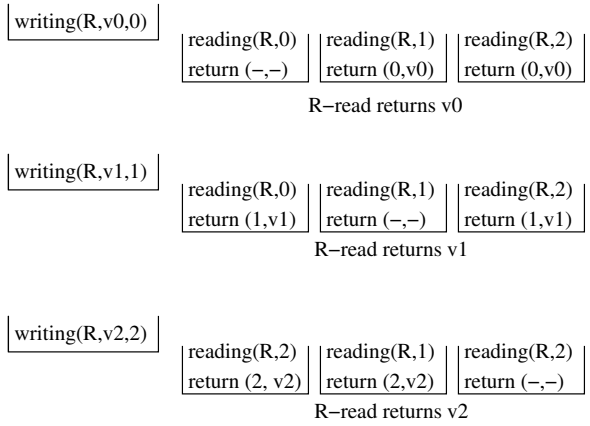

Figure 1: Examples of $\tau 2(\text{RL-READ}(R_{pq}))$ execution without overlapping $\tau 2(\text{RL-WRITE}(R_{pq}, \text{-}))$ execution

4.1 Proof of Compiler 2

Let p and q be two neighbour processors. In this section, all registers are 1W and 1R. When the writer of a register is processor p and the reader is q, we do not indicate the name of the writer or the reader of the register: REG_{pq} is simplely denoted REG.

4.2 Preamble

Definition 4.1

- $st2(i)$ *denote the start time of the ith call of* $\tau 2(\text{RL-WRITE}(R, \text{-}))$. *If the ith call of* $\tau 2(\text{RL-WRITE}(R, \text{-}))$ *does not exist then* $st2(i)$ *has the value* $+\infty$.
- $et2(i)$ *denotes the end time of the ith call of* $\tau 2(\text{RL-WRITE}(R, \text{-}))$ *by processor* p. *If the ith call of* $\tau 2(\text{RL-WRITE}(R, \text{-}))$ *does not exist then* $et2(i)$ *has the value* $+\infty$.
- *We denote by* $et2(-1)$ *the time* 0.
- *If there exists a partial execution of* $\tau 2(\text{RL-WRITE}(R, \text{-}))$ *then* $et2(0)$ *denotes the end time of this partial execution, and we define* $st2(0)$ *has the time* 0.
- *If there does not exist a partial execution of* $\tau 2(\text{RL-WRITE}(R, \text{-}))$ *then* $et2(0)$ *is defined has the time* 0.

Definition 4.2 *Define* $Set(i) = \{Flag[i], R[i \oplus 1], R[i \oplus 2]\}$.

Let $ActR$ *be an execution of reading(*R, c*) by the processor* q.

The ith call of $\tau 2(\text{RL-WRITE}(R, \text{-}))$ *interferes with* $ActR$, *iff* $ActR$ *starts during the time interval* $[st2(i), et2(i))$

We need to show that the value returned by $\tau 2(\text{RL-READ}(R))$ satisfies the semantics of a regular register. The correctness proof has three main steps,

[Step 1] At most one $\tau 2(\text{RL-WRITE}(R, \text{-}))$ can interfere with a given execution of reading(R, c). As a consequence, by the definition of pseudo-regular registers, pseudo-regular registers in $Set(i)$ for any i, satisfy the stronger semantics of regular registers.

[Step 2] The pair of values $(f[i], v[i])$ return by a reading(R, c) execution is the same as the pair of values that would have been computedif reading(R, c) had been executed instantaneously at either (1) the end of the most recent preceding writing(R, -, c') execution where $c' \neq c$ or (2) the end of the interfering $\tau 2(\text{RL-WRITE}(R, \text{-}))$ execution.

[Step 3] The final value returned by $\tau 2(\text{RL-READ}(R))$ is either the value of an overlapping or the most recent preceding $\tau 2(\text{RL-WRITE}(R, \text{-}))$ execution.

Observation 4.1 *Only the registers of* $Set(c)$ *are read by an execution of reading(*R, c*).*

$ActR$ *does not interfere with any* $\tau 2(\text{RL-WRITE}(R, \text{-}))$ *execution iff it exists* $i \geq -1$ *such that* $ActR$ *starts during the time interval* $[et2(i), st2(i+1))$.

writing(R, -, c*) does not write in any register of* $Set(c)$.

Only a writing(R, -, c'*) execution where* $c \neq c'$ *writes in some registers of* $Set(c)$.

Any REG_READING done during an execution of reading(R, c) is overlapped by at most a single REG_WRITING operation, this operation is part of the unique interfering execution of $\tau 2(\text{RL-WRITE}(R, \text{-}))$ (it starts before the starting

time of reading(R, c) and ends after it, we have $c' \neq c$).

Lemma 4.1 *Let $ActR$ be an execution of reading(R, c). Let o be a* PSEUDO-RL-WRITE *operation on a register of $Set(c)$ done by processor p during the execution of $ActR$. Operation o is part of the execution of the $\tau 2$(RL-WRITE(R, -)) interfering with Act.*

Proof: o is done during the execution of writing(R, -, c') where $c \neq c'$ (according to observation 4.1). This execution of writing(R, -, c') is done during an execution of $\tau 2$(RL-WRITE(R, -)) named R-write.

R-write cannot ends before the starting time of $ActR$, and cannot starts after the end of $ActR$.

Assume that R-write starts after or at the starting time of $ActR$. During R-write, only writing(R, -, c) is performed; none register read during $ActR$ is written by R-write (see the Observation 4.1). There is a contradiction. We conclude that R-write starts before the starting time of $ActR$.

R-write is the single interfering $\tau 2$(RL-WRITE(R, -)) execution with $ActR$. □

The previous lemma concludes the first step of the correctness prove. At most one $\tau 2$(RL-WRITE(R, -)) can interfere with a given execution of reading(R, c). As a consequence, by the definition of pseudo-regular registers, pseudo-regular registers in $Set(i)$ for any i, satisfy the stronger semantics of regular registers.

Definition 4.3

- *flag(i, c) denotes the flag value returned by the execution of an instantly reading(R, c) done at time $et2(i)$.*
- *value(i, c) denotes the register value returned by the execution of an instantly reading(R, c) done at time $et2(i)$.*

Observation 4.2 *$\forall i \geq -1$, value(i, c) is the value of register $R[flag(i, c)]_{pq}$ at time $et2(i)$.*

If reading(R, c) execution is not interfered by writing(R, -, c'), then it return the value written by the latest writing(R, -, c') where $c' \neq c$ that happens-before it. And, if a writing(R, -, c') where $c' \neq c$ interfers with reading(R, c) execution, then it returns the value of either the writing(R, -, c') where $c' \neq c$ that happens-before it, or the written value of the overlapping writing(R, -, c') where $c' \neq c$.

Lemma 4.2 *Let $ActR$ be an execution of reading(R, c) by the processor q. If it exists $i \geq -1$ such that $ActR$ starts during the time interval [$et2(i)$,$st2(i+1)$) then $ActR$ returns (flag(i, c), value(i, c)).*

Proof: $ActR$ does not interfere with any any $\tau 2$(RL-WRITE(R, -)) execution (see Observation 4.1). During the execution of $ActR$ no PSEUDO-RL-WRITE operation is done on registers of $Set(c)$ (see Lemma 4.1). The result of $ActR$ is similar at the result of an instantly execution of reading(R, c) at time $et2(i)$. □

Lemma 4.3 *Let $ActR$ be an execution of reading(R, c) by the processor q. If it exists $i \geq 0$ such that $ActR$ starts during the time interval [$st2(i)$,$et2(i)$) then $ActR$ returns (flag(i, c),value(i, c)) or (flag$(i-1, c)$,value$(i-1, c)$).*

Proof: $ActR$ interferes with the ith call of $\tau 2$(RL-WRITE(R, -)) (see definition 4.2).

Let oR be a PSEUDO-RL-READ operation on R (a register of $Set(c)$) done by processor q during the execution of $ActR$. Operation oR is overlapped by at most a single PSEUDO-RL-WRITE operation on R, named oW. Because oW is part of the execution of the ith call of $\tau 2$(RL-WRITE(R, -)) (see Lemma 4.1) and during the execution of ith call of $\tau 2$(RL-WRITE(R, -)), a register of $Set(c)$ is written at most one time. The registers of target systems are pseudo-regular. Thus, oR returns the value in R before the invocation of oW, or the written value by oW.

Let rF be the PSEUDO-RL-READ operation on $Flag[c]$ done by processor q during the execution of $ActR$. rF returns flag$(i-1, c)$ or flag(i, c).

Assume that rF returns flag(i, c). Let us name f the value flag(i, c). At the end of rF, the execution of ith

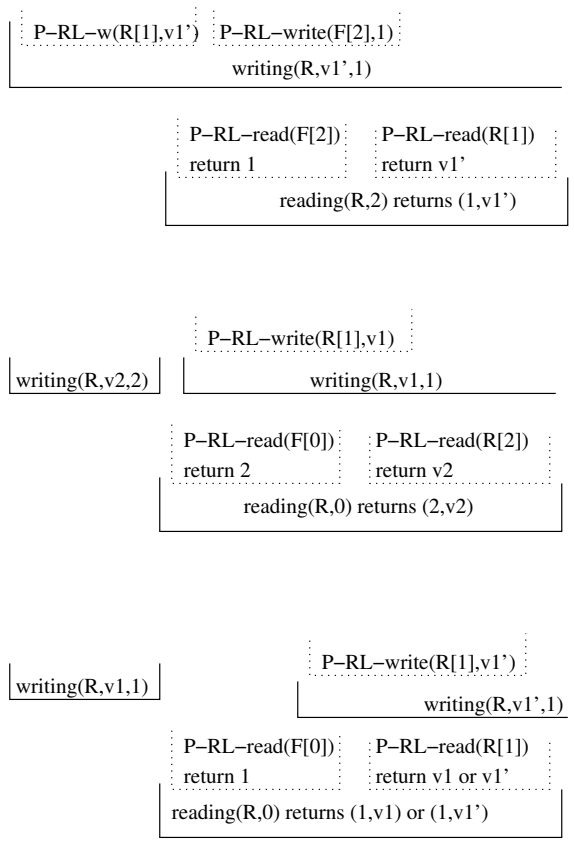

Figure 2: Examples of reading(R, c) execution interfered by a $\tau 2$(RL-WRITE(R, -)) execution

call of $\tau 2$(RL-WRITE(R, -)) is nearly over, the writing operation in $R[f]$ is done. Thus, the PSEUDO-RL-READ operation on $R[f]$ read value(i, c)). $ActR$ returns (flag(i, c), value(i, c)).

Assume that rF returns flag$(i-1, c)$. Let us name f the value flag$(i-1, c)$. If $f \neq$ flag(i, c) then the ith call of $\tau 2$(RL-WRITE(R, -)) does not write in the pseudo-regular register $R[f]$. Thus, the value of $R[f]$ is unchanged, it is value$(i-1, c)$. Thus, $ActR$ returns (flag$(i-1, c)$,value$(i-1, c)$).
If $f ==$ flag(i, c) then the PSEUDO-RL-READ operation on $R[f]$ read value$(i-1, c)$ or value(i, c). Thus, $ActR$ returns (flag(i, c), value(i, c)) or (flag$(i-1, c)$, value$(i-1, c)$). □

The proofs of lemma 4.2 and 4.3 conclude the second step of correctness proof. The pair of values $(f[i], v[i])$ return by a reading(R, c) execution is the same as the pair of values that would have been computed if reading(R, c) had been executed instantaneously at either (1) the end of the most recent preceding writing(R, -, c') execution where $c' \neq c$ or (2) the end of the interfering $\tau 2$(RL-WRITE(R, -)) execution.

4.2.1 Termination

In this section, we prove that any execution (partial or complete) of $\tau 2$(RL-WRITE) and $\tau 2$(RL-READ) terminates.

Lemma 4.4 *Any $\tau 2$(RL-WRITE(R, -)) execution terminates.*

Proof: During the execution of $\tau 2$(RL-WRITE(R, -)), p performs at most three PSEUDO-RL-WRITE and one PSEUDO-RL-READ operations. □

Lemma 4.5 *$\tau 2$(RL-READ(R)) execution terminates.*

Proof: During the execution of $\tau 2$(RL-READ(R)), p performs at most six PSEUDO-RL-READ, three PSEUDO-RL-WRITE, and four internal operations. □

Theorem 4.1 *If $\tau 2$ is a compiler from regular-link model to pseudo-regular-link model then $\tau 2$ is a wait-free compiler.*

Proof: Any $\tau 2$(RL-READ) or $\tau 2$(RL-WRITE) is done in finite number of steps regardless of other processor actions. □

4.2.2 Legitimate Configuration

In this section, we will prove the set of configurations verifying $Leg2$ is an attractor.

Definition 4.4 *Let p and q be two neighbour processors.*
$L1_r(p, q) \equiv$ [p's program_counter is in the pre section of writing(R, -, -)]

$L2_r(p,q) \equiv$ *[p's program_counter is in the post section of writing(*R*, -, -) and* $Flag[c \oplus 2] == c$ *]*
$L3_r(p,q) \equiv$ *[p's program_counter is not in the pre or post section of writing(*R*, -, -) and* $\exists c \in \{0,1,2\}$ *such that* $Flag[c \oplus 2] == Flag[c \oplus 1] == c$ *]*

$Correct\_state2(p,q) \equiv L1_r(p,q) \vee L2_r(p,q) \vee L3_r(p,q)$

$Leg2 \equiv (\forall (p,q) \in E\ Correct\_state2(p,q) \equiv True)$*.*

Lemma 4.6 *Let p and q be two neighbour processors. $Correct\_state2(p,q)$ is closed*

Proof: $L1_r(p,q)$ is verified till p's counter stays in the pre section.

When p's program counter exits of the pre section, we have $Flag[c \oplus 2] == c$ and the p's program counter is in the post section. Thus $L2_r(p,q)$ is verified.

$L2_r(p,q)$ stays verified till p's counter stays in the post section, because the value of $Flag[c \oplus 2]$ is not modified during the post section. When p's program counter exits of the pre section, we have $Flag[c \oplus 2] == Flag[c \oplus 1] == c$ and the p's program counter is not in the pre or post section. Thus $L3_r(p,q)$ is verified if $L2_r(p,q)$ was verified when p's program counter was in the post section.

$L3_r(p,q)$ stays verified till p is not entering in the pre section; because the value of $Flag[.]$ are not modified. □

Lemma 4.7 *Let A be the set of algorithms for the regular-link model that satisfy: every processor p, for any p's neighbour, named q, executes $\tau 2$(RL-WRITE(*R*, -)) at least once after any transient failure. Let $Prot$ be a protocol of A. The set of configuration verifying $Leg2$ is an attractor of target system $T = (G,$ pseudo-regular-link$, \tau 2(Prot))$*

Proof: Let p and q be a pair of neighbour. We need to prove that any execution of T reaches a configuration where $Correct\_state2(p,q)$ is verified for any pair of neighbours.

Let us study the first complete execution of $\tau 2$(RL-WRITE(R, -)) done after a transient failure. Such an execution exists because $Prot$ belongs to A.

When p's program counter is in the pre section, $L1_r(p,q)$ is verified. □

4.2.3 Correctness

Consider a specified system $S = (G,$ regular-link$, Alg)$ where $Alg \in A$. S is transformed by the transformation $\tau 2$ (i.e. Compiler 2) to $T = (G,$ pseudo-regular-link$, \tau 2(Alg))$.

In this section, we will establish that any computation of T from a legitimate configuration, has an interpretation as a computation of S.

Definition 4.5 *If at time $et2(i)$, $L3_r(p,q)$ is verified then it exists $c \in \{0,1,2\}$ $Flag[c \oplus 2] == Flag[c \oplus 1] == c$. c is denoted color(i).*

If at time $et2(i)$, $L3_r(p,q)$ is verified then R[color(i)] is denoted state(i).

Observation 4.3 *The written $value$ during the ith execution of $\tau 2$(RL-WRITE(*R*, -)) is state(i). During the ith execution of $\tau 2$(RL-WRITE(*R*, -)) the only procedure executed is writing(*R*, -, color(i)).*

Then correctness is achieved if (1) any $\tau 2$(RL-READ(R)) that is not overlapped returns the written value of the latest $\tau 2$(RL-WRITE(R, -)) that happens-before it; and if (2) any $\tau 2$(RL-READ(R)) that is overlapped by $\tau 2$(RL-WRITE(R, v')) executions returns the value of either the latest $\tau 1$(PSEUDO-RL-WRITE) on R that happens-before it, or the written value by a overlapping $\tau 2$(RL-WRITE(R, -)) execution. More precisely, The following property have to be proven to establish the correctness. If $Correct\_state2(p,q)$ is verified at $et2(i)$ where $i \geq 1$, then $\forall k \geq 0$,

- any $\tau 2$(RL-READ(R)) starting after or at $et2(i)$ and terminating before $st2(i+k+1)$ retuns state(j) where $j \in [i, i+k]$

Observation 4.4 *If at time $et2(i)$, $Correct\_state2(p, q)$ predicate is verified, we have :*

- *value(i, c) is state(i) if $c \neq$ color(i)*
- *flag(i, c) is color(i) if $c \neq$ color(i)*
- *flag(i, c) is not color(i) if $c =$ color(i)*

Definition 4.6 *Notice $TI(i, k)$ the time interval [$et2(i)$, $st2(i + k + 1)$).*

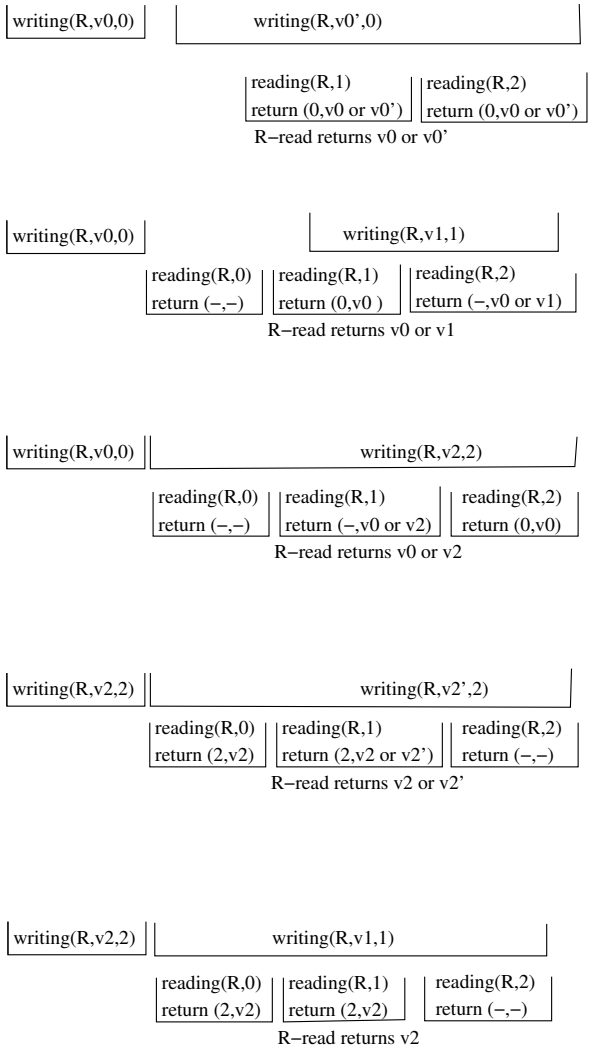

Figure 3: Examples of $\tau 2$(RL-READ(R)) execution with an overlapping $\tau 2$(RL-WRITE(R, -)) execution

Lemma 4.8 *The predicate $Correct\_state2(p, q)$ is verified at time $et2(i)$ where $i \geq 0$.*

Let l be an integer greater than or equal to i. If $\nexists j \in [i, l]$ such that (flag$(l, c1)$, value$(l, c1)$) = (color(j), state(j)) then $c1 =$ color(j) $\forall j \in [i, l]$

Proof: Notice $c2$ the value flag$(l, c1)$. $c2 \neq c1$ (Observation 4.4).

Notice $c3$ the only integer value in {0,1,2} such that $c3 \neq c1$, and such that $c3 \neq c2$.

During $TI(i - 1, l - i)$, no WRITE operation in the pseudo-regular register $R[c2]$ was performed. Otherwise, the value$(l, c1)$ would be state(j) where $j \in [i, l]$ (value$(l, c1)$ is the value of the pseudoregister $R[c2]$ at time $et2(l)$, because $c2$ is the value of the pseudo-register $Flag[c1]$ at time $et2(l)$.) Only, the executions of writing(R, -, c2) include a WRITE operation in the pseudo-regular register $R[c2]$. Thus, during $TI(i, l - i)$, no execution of writing(R, -, c2) was performed.

If during $TI(i - 1, l - i)$, an execution of writing(R, -, c3) was performed then flag$(l, c1)$ would be $c3$, because (1) only the executions of writing(R, -, c') where $c' \neq c1$ include a WRITE operation in the pseudo-regular register $Flag[c1]$, and (2) during $TI(i - 1, l - i)$, no execution of writing(R, -, c2) was performed. Thus, during $TI(i - 1, l - i)$, no execution of writing(R, -, c3) was performed.

During $TI(i - 1, l - i)$, only executions of writing(R, -, c1) was done. According to the code of writing(R, -, c1) $\forall j \in [i, l], c1 =$ color(j). □

Let R-read be an execution of $\tau 2$(RL-READ(R)). Assume that reading(R, c) invocation done during R-read returns a too older value to be acceptable. Then, c is the color of the lastest writing(R, -, -) execution that happen-before R-read call. If $c = 2$ then the flag value returned by the reading(R, 0) invocation and the reading(R, 1) invocation done during R-read is 2. R-read returns the value computed by reading(R, 1). If $c \neq 2$ then the flag value returned by the reading(R, 0) invocation is not equal to the one returned by the reading(R, 1) invocation. R-read call returns the value computed by reading(R, 2).

Theorem 4.2 *Assume that $Correct\_state2(p, q)$ is verified at $et2(i)$. If $i \geq 0$ then any $\tau 2$(RL-READ(R)) starting and terminating during*

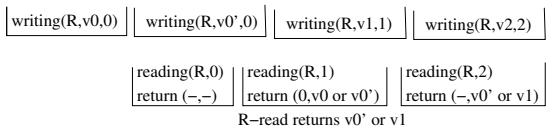

Figure 4: Example of $\tau 2$(RL-READ(R)) execution with several overlapping $\tau 2$(RL-WRITE(R, -)) executions

$TI(i,k)$ *where* $k \geq 0$, *retuns state*(j) *where* $j \in [i, i+k]$.

Proof: Let R-read be an execution of $\tau 2$(RL-READ(R)) starting and terminating during $TI(i,k)$.

Any execution of reading(R, c) done during R-read returns (flag(l,c), value(l,c)) where $l \in [i, i+k]$. Because, this execution starts during $TI(i,k)$ (see lemma 4.2, and lemma 4.3).

For any c value of integer interval $\{0,1,2\}$, we notice l_c the interger value such that the execution of reading(R, c) done during R-read returns (flag(l_c,c), value(l_c,c)). The execution of reading(R, c) done during R-read starts during the time interval [$st2(l_c)$, $et2(l_c+1)$) (see lemma 4.2, and lemma 4.3). We have $l_0 \leq l_1 \leq l_2$ because during R-read, reading(R, 0) is performed before reading(R, 1), and reading(R, 1) is performed before reading(R, 2).

Assume that R-read returns value$(l_2,2)$ and $\nexists j \in [i, i+l_2]$, such that value$(l_2,2)$ = state(j). According lemma 4.8, $\forall j \in [i, i+l_2]$, we have $2 =$ color(j). According to observation 4.4, flag$(l_0,0) =$ $2 =$ color(l_0) and flag$(l_1,1) = 2 =$ color(l_1). In this case, R-read returns value$(l_1,1)$. There is a contradiction.

Assume that R-read returns value$(l_1,1)$ and $\nexists j \in [i, i+l_1]$ such that value$(l_1,1)$ = state(j). According lemma 4.8, $\forall j \in [i, i+l_1]$, we have $1 =$ color(j). According to observation 4.4, flag$(l_0,0)$ $= 1 =$ color(l_0). In this case, R-read returns value$(l_2,2)$. There is a contradiction. □

5 Compiler from regular-link to safe-link

Compiler 3 Code of Self-stabilizing compiler from regular-link networks to safe-link networks

$Flag_1[0..2]_{pq}$, $Flag_2[0..2]_{pq}$, $Flag_3[0..2]_{pq}$, $R_1[0..2]_{pq}$, $R_2[0..2]_{pq}$, $R_3[0..2]_{pq}$, RC_{1qp}, RC_{2qp}, and RC_{3qp} are 1W, 1R safe registers.

$\oplus$ is the addition modulo 3.

Code on the processor p:

τ(RL-WRITE)(R_{pq}, new_state)
  col is a local variable of the procedure.
    col ⟵ REG_READING(RC_{qp});
    REG_WRITING($R[col]_{pq}$, value);
    REG_WRITING($Flag[col \oplus 2]_{pq}$,col);
    REG_WRITING($Flag[col \oplus 1]_{pq}$,col);

τ(RL-READ)(R_{qp})
  $f[0..2]$, $v[0..2]$, and c are local variables.
    for c := 0 to 2 **do**
      REG_WRITING(RC_{pq},c) ;
      $f[c]$ ⟵ REG_READING($Flag[c]_{qp}$);
      if $f[c] \neq c \oplus 1$ **then** f[c] := $c \oplus 2$; **fi**
      $v[c]$ ⟵ REG_READING($R[f[c]]_{qp}$);
    done
    if ($f[0] == f[1] == 2$) **then** return($v[1]$) ;
    else return($v[2]$) ; **fi**

REG_WRITING(REG_{pq}, new_state)
    SL-WRITE(REG_{1pq}, new_state);
    SL-WRITE(REG_{2pq}, new_state);
    SL-WRITE(REG_{3pq}, new_state);

REG_READING(REG_{qp})
  $v1$, $v2$, and $v3$ are local variables of the function.
    $v3$ ⟵SL-READ(REG_{3qp});
    $v2$ ⟵SL-READ(REG_{2qp});
    $v1$ ⟵SL-READ(REG_{1qp});
    if ($v3 == v2$) or ($v1 == v2$) **then** return $v2$;
    else return $v1$; **fi**

Let A be the set of algorithms for the pseudo-regular-link model that satisfy: every processor p, for any p's neighbour, named q, executes τ(RL-WRITE)(R_{pq},-) at least once after any transient failure.

Compiler 3 is the combination of the two previously presented wait-free and stabilizing compilers.

According the the properties of the Compiler 1 and Compiler 2, Compiler 3 is a wait-free and stabilizing compiler from regular-link networks to safe-link networks for all algorithms in A,

6 Conclusion

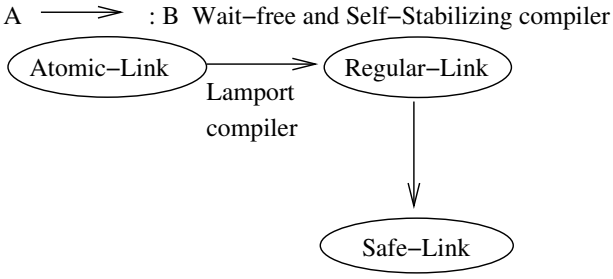

Figure 5: Transformations between link network models

Lamport [10] 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 this relationship between atomic single-writer/single-reader registers and regular single-writer/single-reader registers. Thus, Lamport's implementation constitutes a wait-free compiler, which we call AL-RL, from atomic-link networks to regular-link networks. It is straightforward to confirm that AL-RL is also self-stabilizing.

Using the compiler AL-RL and Compiler 3, self-stabilizing algorithms designed for the atomic-link model could be implemented in the safe-link model in such way that the write and read operations in the target system are wait-free. Many self-stabilizing algorithms are designed for the atomic-link model [5, 3]. Now, these algorithms could be implemented in the safe-link model in such way that the write and read operations in the target system are wait-free.

The known compiler between link model are summarized in the figure 5. The transformation that is not presented in this paper is labelled by the bibliographical reference.

References

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

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

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

[4] S Dolev, MG Gouda, and M Schneider. Memory requirements for silent stabilization. In *PODC96, the 15th Annual ACM Symposium on Principles of Distributed Computing*, pages 27–34, 1996.

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

[6] S Haldar and K Vidyasankar. Constructing 1-writer multireader multivalued atomic variables from regular variables. *Journal of the Association of the Computing Machinery*, 42(1):186–203, 1995.

[7] L Higham and C Johnen. Relationships between communication models in networks using atomic registers. In *IPDPS'06, the 20th IEEE International Parallel & Distributed Processing Symposium*, 2006.

[8] 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.

[9] C. Johnen and L. Higham. Fault-tolerant implementations of atomic-state communication model for distributed computing. In *DISC'07, the 21th International Symposium on Distributed Computing, Springer LNCS:4731*, pages 485–486, 2007. Brief announcement.

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

[11] M Li, J Tromp, and PMB Vitanyi. How to share concurrent wait-free variables. *Journal of the Association of the Computing Machinery*, 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.