# Fault-tolerant Implementations of Regular Registers by Safe Registers with Applications to Networks

Colette Johnen[1] and Lisa Higham[2]

[1] LRI, Univ. Paris-Sud, CNRS, F-91405 Orsay, France * colette@lri.fr
[2] Computer Science Department, University of Calgary, Canada higham@ugalgary.ca

**Abstract.** We present the first wait-free and self-stabilizing implementation of a single-writer/single-reader regular register by single-writer/single-reader safe registers. The construction is in two steps: one implements a regular register using 1-regular registers, and the other implements a 1-regular register using safe-registers. In both steps, if the initial register is bounded then the implementation uses only bounded registers.

## 1 Introduction

Lamport [10] defined three models of single-writer/multi-reader 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, which capture the notion of directly sensing the hardware, are cheaper still. This motivated Lamport and other researchers to find wait-free constructions for assembling atomic registers from regular ones and regular registers from safe ones. One natural extension to this research arises from asking whether these implementations of strong registers from weaker ones can be made self-stabilizing.

This paper addresses this question. The core contribution is an implementation of a single-writer/single-reader regular register using single-writer/single-reader safe registers. The implementation is both self-stabilizing and wait-free. To implement a single-writer/single-reader regular register of $M$ bits, our construction uses 9 single-writer/single-reader safe registers of $M$ bits and 12 single-writer/single-reader safe registers of 2 bits. Thus, if a program uses only bounded regular registers then it can be implemented using bounded safe registers.

**Related research.** A body of research [2, 5, 11, 3], initiated by Lamport [10], gave constructions of strong registers types from weaker ones. By combining these results, even the strongest register being considered (a multi-writer/multi-reader atomic register) has a wait-free construction from a collection of the weakest (single-writer/single-reader safe bits). The fault-tolerance sought in all these constructions was wait-freedom;

* Colette Johnen has moved to Labri, Univ. Bordeaux 1, CNRS, F-33405 Talence, France, johnen@labri.fr

none of these original constructions addressed self-stabilization. More recently, Abraham *et.al.* [1] introduced 1-regular registers, which lie between safe and regular registers, and presented a wait-free implementation of a regular register by 1-regular registers. This construction is also not self-stabilizing and relies on timestamps to distinguish the latest value, thus requiring 1-regular registers of unbounded size.

Hoepman, Papatriantafiou and Tsigas [8] presented self-stabilizing versions of some of these well-known constructions. For instance, they present a wait-free and self-stabilizing implementation of a multi-writer/multi-reader atomic register using single-writer/single-reader regular registers of unbounded size. Dolev and Herman [4] designed a variant of Dijkstra's self-stabilizing token circulation on unidirectional rings that uses only regular registers. In contrast to the wait-free case, however, no self-stabilizing construction of strong registers starting from only safe ones was known.

**Algorithm 1** A binary regular register from a binary safe register

**Shared registers:**

The single-writer/single-reader binary register, $R$, is replaced by a single-writer/single-reader binary safe register, $Rs$.

**Code for writer:**

$\tau$ (REGULAR-WRITE($R$, new_bit)) # $old\_bit$ is a local binary variable #
    if ($old\_bit \neq$ new_bit) then SAFE-WRITE($Rs$, new_bit); $old\_bit \longleftarrow$ new_bit.

**Code for reader:**

$\tau$ (REGULAR-READ($R$)):: return SAFE-READ($Rs$).

Lamport's wait-free implementation of a single-writer/single-reader regular binary register from only one single-writer/single-reader safe binary register (Algorithm 1) is straightforward – the safe register is over-written only if the new value is different from what was most recently written. This simple idea does not work in the self-stabilizing setting, because the value of $old\_bit$ could be corrupted and thus not equal to the value of the safe register. In fact, as established by Hoepman *et.al.* [8], there is no wait-free and self-stabilizing implementation of a single-writer/single-reader regular binary register with only one single-writer/single-reader safe binary register.

**Outline of paper.** The description of our implementation is separated into two pieces. Section 3 presents an implementation of a single-writer/single-reader 1-regular register by single-writer/single-reader safe registers. Section 4 implements a single-writer/single-reader regular register by single-writer/single-reader 1-regular registers. Both of these implementations are wait-free and self-stabilizing; hence, so is their composition. We only sketch the proofs, trying to provide the essential insights for correctness. A lot of detail is omitted, most notably, that needed to completely establish the correctness of the second piece, which is the most intricate of the two. Our technical report [9] has the complete proofs. Section 5 overviews how this work applies to network models of distributed computation, and how it relates to research on fault-tolerant compilers between different variants of network models.

## 2 Definitions and Preliminaries

### 2.1 Model

**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 $o'$ if resp($o$) $<$ inv($o'$). If neither $o$ happens-before $o'$ nor $o'$ happens-before $o$, then $o$ and $o'$ *overlap*. Because there is only one writer, WRITE operations to $R$ happen sequentially, so the happens-before relation is a total order on all the WRITE operations. READ operations, however, may overlap each other and may overlap a WRITE. Lamport [10] defined three kinds of such registers depending on the semantics when READ and WRITE operations overlap. Register $R$ 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 $w$. We also use another register type, which was defined by Abraham *et.al.* [1]. Register $R$ is *1-regular* if it is safe and any READ with *at most one* overlapping WRITE returns the value of either the latest WRITE that happens-before it, or the value of the overlapping WRITE.

**Systems, configurations and executions.** Each of the two technical results of this paper is an implementation of a single-writer/single-reader register (called the *specified register*) using weaker single-writer/single-reader registers (called the *target registers*). As discussed further in Section 5, the implementations extend to any system of such shared registers, because the individual implementations are independent. Thus, the specified system consists of just two processors, a writer $p$ and a reader $q$, sharing a single specified register. This system is implemented using several registers of the target register type. Since our target registers are also single-writer/single-reader, they can be partitioned into two sets. One contains the registers written by $p$ and read by $q$; the other contains the registers read by $p$ and written by $q$.

A *configuration* of the system is an assignment of values to each of the the shared target registers and the internal variables, including the program counter, of the writer and reader implementations. In a *computation step*, a processor executes the next action of its program. Each action is an invocation or a response of an READ or a WRITE on a shared target register, or a read or a write of an internal variable.

### 2.2 Fault-tolerance

**Wait-freedom.** 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. Wait-freedom provides robustness;

it implies that a stopping failure (or very slow execution) of any subset of processors cannot prevent another processor from correctly completing its operation.

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

- $L$ **is attractor:** Starting from any configuration, any computation reaches a configuration satisfying $L$. For any configuration $C$ satisfying $L$, the successor configuration reached by any computation step applied to $C$ also satisfies $L$.
- **correctness from** $L$**:** Any computation starting from a configuration satisfying $L$ satisfies $PS$.

$L$ is called the *legitimacy predicate*. Informally, an algorithm is self-stabilizing to a behaviour specified by $PS$ 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 eventually returns to the specified behaviour.

### 2.3 Proof obligation of register implementations

The possible operations on a register are READ and WRITE.
Therefore, to implement a strong register, $R$, using only weaker registers requires defining two programs $\tau\,(\text{READ}(R))$ and $\tau\,(\text{WRITE}(R, \cdot))$ each accessing only registers of the weaker type. Consider an execution $E$ of the implementation. It consists of a sequence $r_1, r_2, \ldots$ of successive computation steps of $\tau\,(\text{READ}(R))$ by the reader, and a sequence $w_1, w_2, \ldots$ of successive steps of $\tau\,(\text{WRITE}(R, \cdot))$ by the writer. Furthermore, $r_1$ (respectively, $w_1$) could begin part way through $\tau\,(\text{READ}(R))$ (respectively, $\tau\,(\text{WRITE}(R, \cdot))$; there could be arbitrary initial values in registers; and the two sequences of computation steps could overlap arbitrarily.

The proof that an algorithm is a correct wait-free and self-stabilizing implementation of a strong register is decomposed into four components:

**Termination:** Normally, establishing wait-freedom would require showing that any execution of $\tau\,(\text{READ}(R))$ or of $\tau\,(\text{WRITE}(R, \cdot))$ terminates in a finite number of steps. Since our algorithms must simultaneously be self-stabilizing, we need to show the stronger requirement that any execution of any suffix of $\tau\,(\text{READ}(R))$ or of $\tau\,(\text{WRITE}(R, \cdot))$, with any values in the registers, terminates.

**Legitimate configurations:** We define a predicate $\widetilde{L}$ on the values of the target registers and on values of the internal variables used by $\tau\,(\text{WRITE}(R, \cdot))$ and $\tau\,(\text{READ}(R))$.

**Attractor:** We show that $\widetilde{L}$ is an attractor.

**Correctness:** We show that every every execution of $\tau\,(\text{READ}(R))$ that begins from any configuration that verifies the predicate $\widetilde{L}$, returns only values that are consistent with the semantics of the stronger register, $R$.

## 3 Implementing 1-regular Registers from Safe Registers

**Algorithm 2** A 1-regular register constructed from safe registers

**Shared registers:**

Each single-writer/single-reader 1-regular register, $R$, is replaced by 3 single-writer/single-reader safe registers, $R1$, $R2$, $R3$.

**Code for writer:**

$\tau$ (1-REGULAR-WRITE($R$, new_state))
    # line 1: # SAFE-WRITE($R1$, new_state);
    # line 2: # SAFE-WRITE($R2$, new_state);
    # line 3: # SAFE-WRITE($R3$, new_state).

**Code for reader:**

$\tau$ (1-REGULAR-READ($R$))
    # $v1$, $v2$, and $v3$ are local variables of the function. #
    $v3 \longleftarrow$SAFE-READ($R3$); $v2 \longleftarrow$SAFE-READ($R2$); $v1 \longleftarrow$SAFE-READ($R1$);
    if ($v3 = v2$) then return $v2$ else return $v1$.

**Theorem 1.** *Algorithm 2 is a wait-free and self-stabilizing implementation of 1-regular register, R, using safe registers, provided* 1-REGULAR-WRITE*(R, ·) is executed at least once after any transient fault.*

**Termination:** It is immediate from the code that any execution of any suffix of either $\tau$ (1-REGULAR-WRITE) or $\tau$ (1-REGULAR-READ) will terminate since each consists of three read or write operations on safe registers.

**Legitimate configurations:** We define a predicate $\widetilde{Leg1}$ on configurations that (informally) captures the property that there is substantial agreement between the value of the three shared safe registers $R1$, $R2$ and $R3$. This agreement is related to the value of the writer's program counter (denoted PC).

$L1_s \equiv [\ R3 = R2 \wedge$ PC is in line 1 of $\tau$ (1-REGULAR-WRITE($R$, $\cdot$)) $]$
$L2_s \equiv [\ R1 = \nu \wedge$ PC is in line 2 of $\tau$ (1-REGULAR-WRITE($R$, $\nu$)) $]$
$L3_s \equiv [\ R1 = R2 = \nu \wedge$ PC is in line 3 of $\tau$ (1-REGULAR-WRITE($R$, $\nu$)) $]$
$L0_s \equiv [\ R1 = R2 = R3 \wedge$ PC is not in $\tau$ (1-REGULAR-WRITE($R$, $\cdot$)) $]$
$\widetilde{Leg1} \equiv L1_s \vee L2_s \vee L3_s \vee L0_s$.

**Attractor:** Once verified, $L0_s$ remains verified as long as the writer is not executing $\tau$ (1-REGULAR-WRITE($R$, $\nu$)), since registers $R1$, $R2$, and $R3$ are only written inside $\tau$ (1-REGULAR-WRITE($R$, $\nu$)). If $L0_s$ is verified and the PC enters line 1, then $L1_s$ becomes verified because only the value of $R1$ is modified in line 1. When the PC moves to line 2, $R1 = \nu$. Thus, $L2_s$ becomes verified and remains so while PC stays in line 2, because the value of $R1$ is not modified in line 2. When the PC enters line 3, $R1 = R2 = \nu$. Thus $L3_s$ becomes verified and remains so while PC stays in line 3, because only the value of $R3$ is modified. When the PC exits line 3, $R1 = R2 = R3 = \nu$. Thus $L0_s$ is verified. We conclude that $\widetilde{Leg1}$ is closed.

At the end of a complete execution of $\tau$ (1-REGULAR-WRITE($R$, $\cdot$)), $L0_s$ is verified; so $\widetilde{Leg1}$ is verified after one complete execution of $\tau$ (1-REGULAR-WRITE($R$, $\cdot$)).

**Correctness:** Any execution of the transformed system consists of an arbitrary overlapping of two sequences: a sequence of executions of $\tau$ (1-REGULAR-WRITE($R$, $\cdot$)) by the writer and a sequence of executions of $\tau$ (1-REGULAR-READ($R$)) by the reader. Let $E1$ be any such execution that starts from a configuration $c1$ satisfying $\widetilde{Leg1}$. Define *ghost*($R$) to be the value of $R3$ in configuration $c1$.

Let Rd be an execution of $\tau$ (1-REGULAR-READ($R$)) by the reader in $E1$. We need to show that Rd returns a value that could have been returned by the corresponding operation 1-REGULAR-READ($R$) on the original 1-regular register.

If Rd has two or more overlapping executions of $\tau$ (1-REGULAR-WRITE($R$, $\cdot$)) by the writer, then, by the definition of 1-regular registers, any value in the domain of the register can be returned.

Suppose that Rd has at most one overlapping execution of $\tau$ (1-REGULAR-WRITE($R$, $\cdot$)), say W.

Case 1: W was preceded by W-prev, an execution of $\tau$ (1-REGULAR-WRITE($R$, $\cdot$)). Since the three safe registers are accessed in the opposite order by Rd and W, at most one of the safe registers in $\{R1, R2, R3\}$ could be being read by Rd while there is an overlapping write to the same register by W. If the SAFE-READ of $R1$ is overlapped then $v3 = v2$, and the value written by W-prev is returned. If the read of $R3$ is overlapped then $v1 = v2$, and the value written by W is returned. If the read of $R2$ is overlapped then Rd returns either $v2 = v3$ (the value written by W-prev), or $v1$ (the value written by W).

Case 2 : W is the first execution of $\tau$ (1-REGULAR-WRITE($R$, $\cdot$)). This is similar to Case 1, except that Rd returns either the value written by W or the value *ghost*($R$).

Suppose that Rd has no overlapping execution of $\tau$ (1-REGULAR-WRITE($R$, $\cdot$)). Then $L_0$ holds throughout the duration of Rd. So $v3$ is returned; and $v3$ is either the value written by the most recent preceding execution of $\tau$ (1-REGULAR-WRITE($R$, $\cdot$)) or $v3 =$ *ghost*($R$).

# 4 Implementing Regular Registers from 1-regular Registers

## 4.1 Overview of Algorithm 3

If we could ensure that no more than one write could overlap a read operation, a 1-regular register would suffice in place of a regular register. For a single-writer/single-reader model, this observation suggests that we try to avoid overlap by having more than one 1-regular register available for the writer and arranging communication from the reader to direct the writer which one to use. To implement this idea, the regular register $R$ is implemented with three 1-regular copies. There is also a color with value in $\{0, 1, 2\}$ associated with $R$. The color value is written by the reader and read by the writer (also using a 1-regular register). The writer implements a REGULAR-WRITE to $R$ by writing to the copy $R[i]$ if it believes the current color is $i$. Three additional

1-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.

**Algorithm 3** A regular register constructed from 1-regular registers

**Shared registers:**

The single-writer/single-reader regular register, $R$ is replaced by 7 single-writer/single-reader 1-regular registers: $R[c]$ and $Flag[c]$ $\forall c \in [0..2]$ each written by the writer and read by the reader, and $RC$ written by the reader and read by the writer.

$\oplus$ denotes addition modulo 3.

**Code for writer:**

$\tau$ (REGULAR-WRITE($R$, $new\_state$)) # color is a local variable of the procedure. #

```
    # line 1: #   color ⟵ 1-REGULAR-READ(RC);
                  1-REGULAR-WRITE(R[color], new_state);
    # line 2: #   if (color ≠ 2) then 1-REGULAR-WRITE(Flag[2], color);
    # line 3: #   if (color ≠ 1) then 1-REGULAR-WRITE(Flag[1], color);
    # line 4: #   if (color ≠ 0) then 1-REGULAR-WRITE(Flag[0], color).
```

**Code for reader:**

$\tau$ (REGULAR-READ($R$)) # $f[0..2]$, $v[0..2]$, and $c$ are local variables. #

```
    for c ⟵ 0 to 2 do
        1-REGULAR-WRITE(RC, c);
        f[c] ⟵ 1-REGULAR-READ(Flag[c]);
        if f[c] ≠ c ⊕ 1 then f[c] ⟵ c ⊕ 2;
        v[c] ⟵ 1-REGULAR-READ(R[f[c]]);
    if (f[0] = f[1] = 2) then return(v[1]) else return(v[2]).
```

Consider a regular register with writer $p$ and reader $q$. In a $\tau$ (REGULAR-WRITE($R$, $\cdot$)) execution, $p$ first reads $RC$ to get a color $i \in \{0, 1, 2\}$. It then writes its new state to $R[i]$, and set both registers $Flag[i \oplus 2]$ and $Flag[i \oplus 1]$ to $i$ thus making them "point to" the register just written.

In an execution of $\tau$ (REGULAR-READ($R$)), the reader $q$ executes a loop three times. In iteration $i \in \{0, 1, 2\}$ of the loop, the reader $q$ first writes $i$ to $RC$, and then reads $Flag[i]$ to get a pointer value. Depending on the value returned, $q$ reads either $R[i \oplus 1]$ or $R[i \oplus 2]$. Thus the reader $q$ gets a pair of values $f[i], v[i]$ in iteration $i$. Reader $q$ returns either $v[1]$ or $v[2]$ depending on the flag values read during iterations 0 and 1.

### 4.2 Sketch of Algorithm 3 Correctness

**Theorem 2.** *Algorithm 3 is a wait-free and self-stabilizing implementation of regular register, R, using 1-regular registers, provided* REGULAR-WRITE*(R, ·) is executed at least once after any transient fault.*

**Termination:** It is immediate from the code that any execution of any suffix of either $\tau$ (REGULAR-WRITE($R$, $\cdot$)) or $\tau$ (REGULAR-READ($R$)) will terminate since each consists of nine or fewer read or write operations on 1-regular registers.

**Legitimate Configurations:** Observe that the local variables are overwritten with values that do not depend on their previous values. This leads to the intuition that, after each is overwritten, the configuration is the same as one that could arise from a complete fault-free execution. Define a legitimacy predicate $\widetilde{Leg2}$ on the configurations as follows.

$L1_r \equiv$ [ PC is in line 1 or is not in any line of $\tau$ (REGULAR-WRITE($R$, $\cdot$))
$\wedge \exists$ c $\in \{0,1,2\}$ satisfying $Flag[c \oplus 2]$ = $Flag[c \oplus 1]$ = c ]

$L2_r \equiv$ [ PC is in line 2 of $\tau$ (REGULAR-WRITE($R$, $\nu$)) $\wedge$ $R[color] = \nu$ ]

$L3_r \equiv$ [ PC is in line 3 of $\tau$ (REGULAR-WRITE($R$, $\nu$)) $\wedge$ $R[color] = \nu$ $\wedge$
$((Flag[2] = color) \vee (2 = color))$ ]

$L4_r \equiv$ [ PC is in line 4 of $\tau$ (REGULAR-WRITE($R$, $\nu$)) $\wedge$ $R[color] = \nu$ $\wedge$
$[(Flag[2] = color = 0 = Flag[1]) \vee (Flag[2] = color = 1) \vee$
$(Flag[1] = color = 2)]$ ]

$\widetilde{Leg2} \equiv L1_r \vee L2_r \vee L3_r \vee L4_r$

**Attractor:** It is easily confirmed from the code that $\widetilde{Leg2}$ is verified after one complete execution of $\tau$ (REGULAR-WRITE($R$, $\cdot$)) and that once it is verified it remains verified. So $\widetilde{Leg2}$ is an attractor.

**Correctness:** We start with some notation to facilitate the correctness argument. Let Rd denote an execution by the reader of $\tau$ (REGULAR-READ($R$)) starting from a legitimate configuration. Denote by Rd-loop($i$) the interval within the execution of Rd when the reader is executing iteration $i$ of its loop, for $i \in \{0, 1, 2\}$. Let W denote an execution of $\tau$ (REGULAR-WRITE($R$, $\cdot$)). Say that W *writes with color* $j$ if the operation 1-REGULAR-READ($RC$) done during the execution of W returns $j$.

Consider a computation $E2$ starting from a configuration $c2$ satisfying $\widetilde{Leg2}$. Rather than treat the starting point $c2$ separately, we define a *ghost* W that writes with a color and value defined by the values of the 1-regular registers in $c2$. Specifically, if in configuration $c2$, there is $c \in \{0, 1, 2\}$ satisfying $Flag[c \oplus 2] = Flag[c \oplus 1] = c$ then define initialize_$value = R[c]$ and define initialize_$color = c$. Otherwise define initialize_$value = R[c']$ where $c' = Flag[2]$, and define initialize_$color = c'$. The *ghost* W writes the value initialize_$value$ with color initialize_$color$.

Define $Set.i$ = { $Flag[i]$, $R[i \oplus 1]$, $R[i \oplus 2]$ }. Say that W *interferes* with Rd-loop($i$) only if it both overlaps with Rd-loop($i$) and it writes to at least one 1-regular register that is read by the reader during Rd-loop($i$).

We have to prove that a $\tau$ (REGULAR-READ($R$)) satisfies the requirement of a regular register. The proof has three main steps.

**Step 1** At most one W can interfere with a given Rd-loop($i$). As a consequence, by the definition of 1-regular registers, 1-regular registers in $Set.i$ for any $i$, satisfy the stronger semantics of regular registers.

**Step 2** The pair of values $(f[i], v[i])$ returned by Rd-loop($i$) is the same as the pair of values that would have been computed had Rd-loop($i$) been executed instantaneously either (1) at the end of the most recent preceding W, or (2) at the end of the interfering W.

**Step 3** The final value returned by Rd is either the value of an overlapping or the most recent preceding W.

**Step 1:** Observe from the code that Rd-loop($i$) reads only from registers in $Set.i$, and if the writer writes with color $i$, it does not access any registers in $Set.i$. Thus, to interfere with Rd-loop($i$), W must write with color $j \neq i$, implying it must begin 1-REGULAR-READ($RC$) before Rd-loop($i$) completes 1-REGULAR-WRITE($RC$, $i$). Also, it must overlap at least the first read, 1-REGULAR-READ($Flag[i]$), by Rd-loop($i$). Because there is only one writer, there can be is at most one such W that spans this interval. In the Fig. 1, W (writing with the color 2) interferes with Rd-loop($0$).

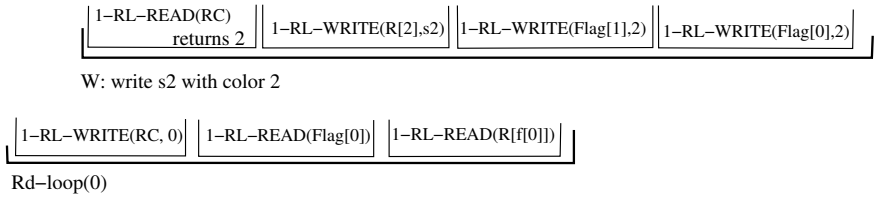


**Fig. 1.** Illustration of step 1

**Step 2:** If no W interferes with Rd-loop($i$), then by the definition of 1-regular registers, the pair $(f[i], v[i])$ computed in Rd-loop($i$), will be the same as would be computed had Rd-loop($i$) been executed any time after the registers it accesses were last written, which is no later than at the end of the most recent preceding W. Suppose, that some W interferes with Rd-loop($i$) and let W-prev denote the $\tau$ (REGULAR-WRITE($R$, $\cdot$)) that most recently precedes W. Notice that W first writes to the $R$ 1-regular register and then to the $Flag$ 1-regular registers, whereas Rd-loop($i$) first reads a $Flag$ 1-register and then an $R$ 1-register. Suppose W writes with color $k$ ($k \neq i$). The value $f[i]$ is either the value, say $j$, in $Flag[i]$ at the end of W-prev, or the value $k$ written by W to $Flag[i]$. Figure 2 illustrates this, for the case where $j = 2$ and $k = 1$. If $f[i] = j$ and $j \neq k$ then the read of $R[j]$ in Rd-loop($i$) returns the value of $R[j]$ at the end of W-prev because W does not write to $R[j]$. If $f[i] = k$ and $j \neq k$, then the read of $R[k]$ in Rd-loop($i$) will return the value just written by W. If $f[i] = j = k$, then the read of $R[k]$ in Rd-loop($i$) will return either the value written by W or by W-prev. In all cases, the pair of values $(f[i], v[i])$ is the pair of values from either the end of W-prev or the end of W.

**Step 3:** Consider Rd-loop($c$) of Rd, for some $c \in \{0, 1, 2\}$. The value, $v[c]$, computed by Rd-loop($c$) is the value written by a real or ghost $\tau$ (REGULAR-WRITE($R$, $\cdot$)) execution named W_stale or it is the initial value of $R[f[c]]$. We denoted by $t0$ the ending time of W_stale if W_stale exists otherwise $t0$ denoted the time $t - 1$ - assuming that the ghost $\tau$ (REGULAR-WRITE($R$, initialize_$value$)) execution ends at time $t$. Let W_previous denote the latest execution of $\tau$ (REGULAR-WRITE($R$, $\cdot$)) preceding Rd, and $t1$ be its ending time. We will show that Rd will not return $v[c]$ as its final value if

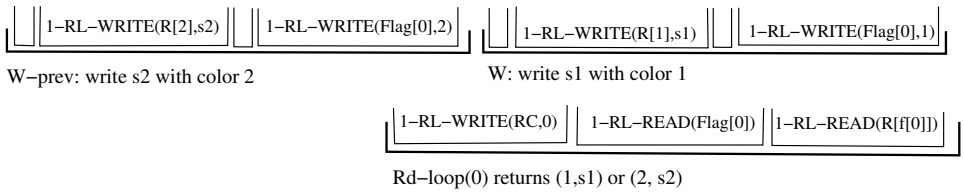


**Fig. 2.** Illustration of step 2

$t0 < t1$. It will therefore follow that the final value returned by Rd is either the value of an overlapping or the most recent preceding $\tau$ (REGULAR-WRITE($R$, $\cdot$)) execution.

Let $I(c)$ denote the interval from $t0$ to the beginning of Rd-loop($c$).

$R[f[c]]$ has not been overwritten during interval $I(c)$. Any write with color $f[c]$ writes to $R[f[c]]$. Thus, all writes during $I(c)$ do not have color $f[c]$. According to Rd-loop($c$) property, the flag value returned by Rd-loop($c$) is the the value of $f[c]$ at $t1$ or at $t2$ (where $t2$ is the end of the interfering W with Rd-loop($c$)). Observe that if a write with color $j$ such that $c \neq j \neq f[c]$ is done during $I(c)$ then $f[c]$ would be equal to $j$ at $t1$ and at $t2$. Thus, all writes during $I(c)$ have color $c$.

Observe from the code, that the final value returned by Rd is either $v[1]$ (if $f[0] = f[1] = 2$) or $v[2]$ (otherwise). Assume that $c > 0$; let $k$ be an integer such that $0 \leq k < c$. Notice that the time interval $I(k)$ is a prefix of $I(c)$. If $t0 < t1$ then a write with color $c$ (ghost or real) occurs during $I(k)$.

Rd-loop($k$) property implies that Rd-loop($k$) computes $f[k] = c$. If $c = 2$ then Rd returns $v[1]$ ($f[0] = f[1] = 2$) otherwise Rd returns $v[2]$ ($f[0] = 1$).

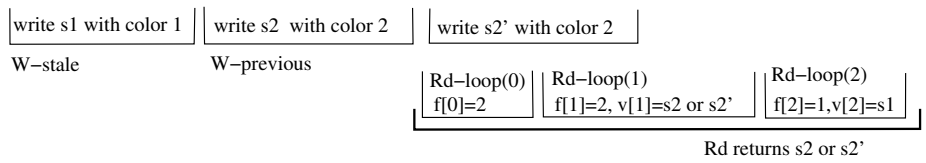


**Fig. 3.** Illustration of step 3.

We conclude that the final value returned by Rd is either the value of an overlapping or the most recent preceding $\tau$ (REGULAR-WRITE($R$, $\cdot$)) execution.

In Figure 4, a sequence of three $\tau$ (REGULAR-WRITE($R$, $\cdot$)) executions is concurrent with a $\tau$ (REGULAR-READ($R$)).

The possible values returned by the $\tau$ (REGULAR-READ($R$)) execution are indicated.

**Combining the implementations.** The composition of Algorithms 2 and 3 is a self-stabilizing implementation of a single-writer/single-reader regular register using only single-writer/single-reader safe registers. Also observe that both implementations are

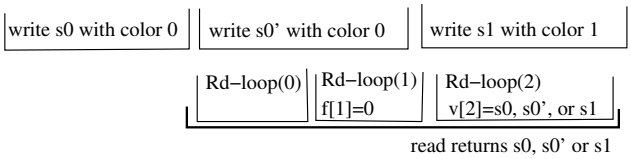


**Fig. 4.** Example of an execution

clearly wait-free — each transformation is straight-line code of at most 10 operations on shared registers. Hence, their composition is also wait-free. The size of each shared safe register used by the composition is $9M + 24$ bits where $M$ is the size of the regular register being implemented. Thus, any self-stabilizing algorithm that uses only bounded single-writer/single-reader regular registers, can be implemented with bounded single-writer/single-reader safe registers.

## 5 Application to Network Simulations

A network that uses shared registers can be modelled as 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 single-writer/multi-reader and located at the nodes (*state* models) or single-writer/ single-reader and located on the edges (*link* models). By specifying either state or link communication, via shared registers that are either safe, regular, or atomic, we arrive at six different register-based network models. For a graph $G$, strength-location$(G)$ denotes the network with topology $G$ and network model strength-location, where strength $\in \{$safe, regular, atomic$\}$ and where location $\in \{$link, state$\}$. For example, the regular-link model has single-writer/single-reader regular registers located on the links of the network.

In a network model, two processors can share a register only if they share a edge in $G$. In contrast, in the stronger globally shared memory model, even when the globally shared memory contains only single-writer/single-reader registers, *any* pair of processors can share registers.

The research cited earlier (Lamport and others) on wait-free implementations of strong register types using weaker ones exploited globally shared memory for some of the implementations of multi-reader (respectively, multi-writer) registers using only single-reader (respectively, single-writer) registers. A natural generalization of this research is to determine which of these implementations remain possible in the register-based network model.

Our research seeks to answer this question as well as to generalize by adding self-stabilization to the fault-tolerance requirement. Specifically, we ask whether it is possible to transform wait-free (respectively, self-stabilizing) algorithms for one of the stronger network models into a wait-free (respectively, self-stabilizing) algorithms for a weaker network model.

The implementations in this paper immediately answer one part of this question: there is a wait-free and self-stabilizing implementation of algorithms designed for regular-link($G$) on safe-link($G$), for any topology $G$.

By combining our previous work with either existing results or straightforward extensions of known results, we can also answer all parts of the question concerning conversions from atomic to regular strength. Specifically, 1) Previous research [7] presents a self-stabilizing implementation of atomic-state($G$) on regular-state($G$), and proves that no such wait-free implementation exists [6]. 2) Lamport (construction 5 of [10]) gave a wait-free implementation of an atomic single-writer/single-reader register by regular single-writer/single-reader registers. It is easily confirmed that this implementation is also self-stabilizing, and constitutes a wait-free and self-stabilizing implementation of atomic-link($G$) on regular-link($G$). 3) A natural way to transform an algorithm from a state model into an algorithm for a link model is to implement a WRITE by sequentially writing to each adjacent link. While this algorithm fails to preserve atomicity, it suffices for regular (or safe) registers. It is straightforward to confirm that this naive idea provides a wait-free and self-stabilizing implementation of regular-state($G$) on regular-link($G$).

Given these previously known results and the contributions of this paper, the only remaining open piece of the general question is whether or not there is a wait-free and/or self-stabilizing implementation of regular-state($G$) on safe-state($G$) for any graph $G$.

## References


1. I Abraham, G Chockler, I Keidar, and D Malkhi. Wait-free regular storage from byzantine components. *Information Processing Letters*, 101(2):60–65, 2007.
2. U Abraham. On interprocess communication and the implementation of multi-writer atomic registers. *Theoretical Computer Science*, 149(2):257–298, 1995.
3. H Attiya and JL Welch. *Distributed computing: fundamentals, simulations and advanced topics*. McGraw-Hill, Inc., 1998.
4. S Dolev and T Herman. Dijkstra's self-stabilizing algorithm in unsupportive environments. In *WSS01, the 5th International Workshop on Self-Stabilizing Systems, Springer LNCS:2194*, pages 67–81, 2001.
5. 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.
6. L Higham and C Johnen. Relationships between communication models in networks using atomic registers. In *IPDPS'2006, the 20th IEEE International Parallel & Distributed Processing Symposium*, 2006.
7. L Higham and C Johnen. Self-stabilizing implementation of atomic register by regular register in networks framework. Technical Report 1449, L.R.I., 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. Self-stabilizing implementation of regular register by safe registers in link model. Technical Report 1486, L.R.I, 2008.
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.