Self-Stabilization of Wait-Free Shared Memory Objects*

Jaap-Henk Hoepman[1], Marina Papatriantafilou[2,3], and Philippas Tsigas[3]

[1] CWI, P.O. Box 94079, 1090 SB Amsterdam, The Netherlands.
[2] CTI & CE and Informatics Dept., Patras University, Greece.
[3] MPI für Informatik, Im Stadtwald, 66123 Saarbrücken, Germany.
Email: jhh@cwi.nl, {ptrianta,tsigas}@mpi-sb.mpg.de

Abstract. Past research on fault tolerant distributed systems has focussed either on processor failures, ranging from benign crash failures to the malicious Byzantine failure types, or on transient memory errors, which can suddenly corrupt the state of the system. It is an interesting question whether one can device highly fault tolerant distributed protocols that tolerate both processor failures as well as transient memory errors. To answer this question we consider self-stabilizing wait-free shared memory objects. These objects occur naturally in distributed systems in which both processors and memory may be faulty. Our contribution in this paper is threefold. First, we propose a general definition of a self-stabilizing wait-free shared memory object that expresses safety guarantees even in the face of processor failures. We prove that within this framework one cannot construct a self-stabilizing single-reader single-writer regular bit from single-reader single-writer safe bits—which have traditionally been used as the basic building blocks in wait-free shared register implementations. This impossibility result leads us to postulate a self-stabilizing *dual*-reader single-writer safe bit as the minimal object needed to achieve self-stabilizing wait-free interprocess communication and synchronization. Finally, we formally prove that, based on this model, adaptations of well known wait-free implementations of regular and atomic shared registers are self-stabilizing.

1 Introduction

The importance of reliable distributed systems can hardly be exaggerated. In the past, research on fault tolerant distributed systems has focused either on system models in which processors fail, or on system models in which the memory is faulty. In the first model a distributed system must remain operational while a certain fraction of the processors is malfunctioning. When constructing shared

* Research partially supported by the Dutch foundation for scientific research (NWO) through NFI Proj. ALADDIN (contr. # NF 62-376) and a NUFFIC Fellowship, and by the EC ESPRIT II BRA Proj. ALCOM II (contr. # 7141). This work will appear in the Proceedings of the 9th International Workshop on Distributed Algorithms (WDAG'95), Lecture Notes in Computer Science, Springer-Verlag, 1995.

memory objects like, for instance, atomic registers, this issue is addressed by considering *wait-free* constructions which guarantee that any operation executed by a single processor is able to complete even if all other processors crash in the meantime. Originally, research in this area focussed on the construction of atomic registers from weaker (safe or regular) ones [VA86, Lam86, PB87, LTV89, IS92]. Later attention shifted to stronger objects (cf. [AH90, Her91] and many others).

In the second model a distributed system is required to overcome arbitrary changes to its state within a bounded amount of time. If the system is able to do so, it is called *self-stabilizing*. Self-stabilizing protocols have been extensively studied in the past. Originating from the work of Dijkstra on self-stabilizing mutual exclusion on rings [Dij74], several other self-stabilizing protocols have been proposed for particular problems, like mutual exclusion on other topologies [BGW89, BP89, DIM93], the construction of a spanning-tree [AKY90], orienting a ring [IJ93, Hoe94], and network synchronization [AKM$^+$93]. Another approach focuses on the construction of a 'compiler' to automatically transform a protocol belonging to a certain class to a similar, self-stabilizing, one [KP90, AKM$^+$93]. For a general introduction to self-stabilization see [Sch93, Tel94].

To develop truly reliable systems both failure models must be considered together. We briefly summarize recent theoretical research that addresses this issue. Anagnostou and Hadzilacos [AH93] show that no self-stabilizing, fault-tolerant, protocol exists to determine, even approximately, the size of a ring. Gopal and Perry [GP93] present a 'compiler' to turn a fault-tolerant protocol for the synchronous rounds message-passing model into a protocol for the same model which is both fault-tolerant and self-stabilizing. A combination of self-stabilization and wait-freedom in the construction of clock-synchronization protocols is presented in [DW93, PT94]. Another approach to combining processor and memory failures is put forward by Afek et al. [AGMT92, AMT93] and Jayanti et al. [JCT92]. They analyze whether shared objects do or do not have wait-free (self)-implementations from other objects of which at most t are assumed to fail. Objects may fail by giving responses which are incorrect, or by responding with a special error value, or even by not responding at all. In so-called gracefully degrading constructions, operations during which more than t objects fail are required to fail in the same manner.

We are interested in exploring the relation between self-stabilization and wait-freedom in shared memory objects. A shared memory object is a data structure stored in shared memory which may be accessed concurrently by several processors through the invocation of operations defined for it. Self-stabilizing wait-free objects occur naturally in distributed systems in which both processors and memory may be faulty. We give a general definition of self-stabilizing wait-free shared memory objects, and focus on studying the self-stabilizing properties of wait-free shared registers. Single-writer single-reader safe bits—traditionally used as the elementary memory units to build these registers with—are shown to be too weak for our purposes. Focusing on registers, being the weakest type of shared memory objects, allows us to determine the minimal object properties needed for a system to be able to converge to legal behaviors

after transient memory faults, as well as to remain operative in the presence of processor crashes.

Shared registers are shared objects reminiscent of ordinary variables, that can be read or written by different processors concurrently. They are distinguished by the level of consistency guaranteed in the presence of concurrent operations ([Lam86]). A register is *safe* if a read returns the most recently written value, unless the read is concurrent with a write in which case it may return an arbitrary value. A register is *regular* if a read returns the value written by a concurrent or an immediately preceding write. A register is *atomic* if all operations on the register appear to take effect instantaneously and act consistent with a sequential execution. Shared registers are also distinguished by the number of processors that may invoke a read or a write operation, and by the number of values they may assume. These dimensions imply a hierarchy with single-writer single-reader ($1W1R$) binary safe registers (a.k.a. bits) on the lowest level, and multi-writer multi-reader ($nWnR$) l-ary atomic registers on the highest level. A *construction* or *implementation* of a register is comprised of i) a data structure consisting of memory cells called *sub-registers* and ii) a set of read and write procedures which provide the means to access it.

Li and Vitányi [LV91] and Israeli and Shaham [IS92] were the first to consider self-stabilization in the context of shared memory constructions. Both papers implicitly call a shared memory construction self-stabilizing if for every *fair* run started in an arbitrary state, the object behaves according to its specification except for a finite prefix of the run. Moreover, they do not seem to consider the possibility of pending operations. We feel, however, that this notion of a self-stabilizing object does not agree well with the additional requirement that the object is wait-free, since self-stabilization of the object now only guarantees recovery from transient errors in fair runs (in which no processors crash), while an object should be wait-free to ensure that a single processor can make progress even if all other processors have crashed.

Our contribution in this paper is threefold. First, in Sect. 2, we propose a general definition of a self-stabilizing wait-free shared memory object, that ensures that all operations after a transient error will eventually behave according to their specification even in the face of processor failures. Second, in Sect. 3, we prove that within this framework one cannot construct a self-stabilizing single-reader single-writer regular bit from single-reader single-writer safe bits—which have traditionally been used as the basic building blocks in wait-free shared register implementations. This impossibility result leads us to postulate a self-stabilizing *dual*-reader single-writer safe bit, which models a flip-flop with its output wire split in two (cf. Sect. 4). Using this bit as a basic building block, we formally prove, as a third contribution, that adaptations of well known wait-free implementations of regular and atomic shared registers are self-stabilizing (cf. Sects. 4.1, 4.2, and 4.3). This shows that our definition of self-stabilizing wait-free shared objects is viable—in the sense that it is neither trivial nor impractical. Section 5 concludes this paper with directions for further research.

2 Defining Self-Stabilizing Wait-Free Objects

In the definition of shared memory objects we follow the concept of *linearizability* (cf. [Her91]), which we, for the sake of self containment, briefly paraphrase here. Consider a distributed system of n sequential processors. A shared memory object is a data-structure stored in shared memory that may be accessed by several processors concurrently. Such an object defines a set of *operations* $\mathcal{O}$ which provide the only means for a processor to modify or inquire the state of the object. The set of processors that can invoke a certain operation may be restricted. Each operation $O \in \mathcal{O}$ takes zero or more parameters p on its invocation and returns a value r as its response ($r = O(p)$). Each such operation execution is called an *action* and is a sequential execution of a procedure's steps; each step may be either a *sub-operation* on the cells of the data structure, or local computations of the procedure. We denote by $t_i(A) \geq 0$ the invocation time of an action A and by $t_r(A) > t_i(A)$ its response time (on the real time axis). Processors are sequential and, therefore, cannot invoke an action if their previously invoked action has not responded yet. To model processor crash failures we introduce for each processor p a *crash action* ψ_p. No invocation or response of an action at processor p, nor another crash action ψ_p may occur later than the time $t(\psi_p)$ processor p crashes. In terms of the implementation of an object, no sub-operations may be executed by p after the crash action ψ_p either.

The desired behavior of an object is described by its *sequential specification* $\mathcal{S}$. This specifies the state of the object, and for each operation its effect on the state and its (optional) response. We write $(s, r = O(p), s') \in \mathcal{S}$ if invoking O with parameters p in state s changes the state of the object to s' and returns r as its response. A *run* over the object is a tuple $\langle \mathcal{A}, \rightarrow \rangle$ with actions $\mathcal{A}$ and partial order $\rightarrow$ such that for $A, B \in \mathcal{A}$, $A \rightarrow B$ iff $t_r(A) < t_i(B)$. Similarly, $\psi_p \rightarrow A$ iff $t(\psi_p) < t_i(A)$. If two actions are incomparable under $\rightarrow$, they are said to *overlap*. Runs have infinite length, and capture the real time ordering between actions invoked by the processors. An implementation of a shared object is *wait-free* if in all runs each invocation of an action A is followed by a matching response after finite time (i.e. $t_r(A) - t_i(A) < \infty$), unless the processor p invoking A crashed at time $t(\psi_p)$ such that $0 < t(\psi_p) - t_i(A) < \infty$. A *sequential execution* $\langle \mathcal{A}, \Rightarrow \rangle$ over the object is an infinite sequence $s_1 A_1 s_2 A_2 \ldots$, where $\bigcup_i A_i = \mathcal{A}$, s_i a state of the object as in its sequential specification, and $\Rightarrow$ a total order over $\mathcal{A}$ defined by $A_i \Rightarrow A_j$ iff $i < j$. A run $\langle \mathcal{A}, \rightarrow \rangle$ *corresponds* with a sequential execution $\langle \mathcal{A}, \Rightarrow \rangle$ if the set of actions $\mathcal{A}$ is the same in both runs, and if $\Rightarrow$ is a total extension of $\rightarrow$ (i.e. $A \rightarrow B$ implies $A \Rightarrow B$). Stated differently, the sequential execution corresponding to a run is a run in which no two actions are concurrent but in which the 'observable' order of actions in the run is preserved.

Definition 1. A run $\langle \mathcal{A}, \rightarrow \rangle$ over an object is *linearizable w.r.t. sequential specification* $\mathcal{S}$, if there exists a corresponding sequential execution $\langle \mathcal{A}, \Rightarrow \rangle$, such that $(s_i, A_i, s_{i+1}) \in \mathcal{S}$ for all i.

An object is linearizable w.r.t. its sequential specification $\mathcal{S}$ if all possible runs over the object are linearizable w.r.t. $\mathcal{S}$. Informally speaking, an object is lineari-

zable w.r.t.to specification $\mathcal{S}$ if all actions appear to take effect instantaneously and act according to $\mathcal{S}$.

2.1 Adding Self-Stabilization

Li and Vitányi [LV91] and Israeli and Shaham [IS92] were the first to consider self-stabilizing wait-free constructions. Both papers implicitly use the following straightforward definition of a self-stabilizing wait-free object.

Definition 2. A shared wait-free object is self-stabilizing if an arbitrary *fair* execution (in which all operations on all processors are executed infinitely often) started in an arbitrary state, is linearizable except for a finite prefix.

A moment of reflection shows that assuming fairness may not be very reasonable for wait-free shared objects. The above definition requires that after a transient error *all* processors cooperate to repair the fault. On the other hand, wait-freedom should imply that processors can make sensible progress even if other processors have crashed. This observation leads us to the following stronger, still informal, definition of a self-stabilizing wait-free shared object.

Definition 3. A shared wait-free object is self-stabilizing, if an arbitrary execution started in an arbitrary state is linearizable except for a bounded finite prefix.

Let us develop a formal version of this definition. To model self-stabilization we need to allow runs that start in an arbitrary state; in particular we have to allow runs in which a subset of the processors start executing an action at an arbitrary point within its implementation. Such runs model the case in which transient memory errors occurs during an action, or, rather, the case where alteration of the program counter by the transient error forces the processor to jump to an arbitrary point within the procedure implementing the operation. For such so called *pending* actions A, and for such actions alone, we set $t_i(A) = 0$. Slow pending actions can carry the effects of a transient error arbitrarily far into the future[4]. Hence we can only say something meaningful about that part of a run after the time that all pending actions have finished, or the processors on which these pending actions run have crashed.

An action A *overlaps a pending action* iff there exists a pending action B on some processor p with $t_i(B) = 0$ such that $t_r(B) \not< t_i(A)$ and $t(\psi_p) \not< t_i(A)$. Define $\mathsf{count}(A)$ equal 0 for all actions A overlapping a pending action, and define $\mathsf{count}(A)$ equal to i if A is executed as the i-th action of a certain processor not

[4] A pending action may carry something "malicious" in its local state that might disorder the system at any time after it is used to modify a sub-register. Consider for instance the Vitányi-Awerbuch register (cf. [VA86] and Sect. 4.3). If a pending action writes a huge tag only to the last register S_{in} in the row, no later action except one executed by processor n will see this tag. If writes after the pending action use a lower tag, they will be ignored by actions of processor n, even if these writes occur strictly before the actions of processor n.

overlapping a pending action. As a special case—to be used later—for all actions A with $\mathsf{count}(A) = 0$ for which there exists a B with $\mathsf{count}(B) = 1$ and $A \| B$ set $\mathsf{count}(A) = \oplus$ instead. Actions A with $\mathsf{count}(A) = \oplus$ overlap both with pending actions and with actions not overlapping any pending actions. As each processor executes sequentially, and actions are unique, count is well-defined.

Definition 4. A run $\langle \mathcal{A}, \rightarrow \rangle$ is *linearizable w.r.t. sequential specification* $\mathcal{S}$ *after* k *processor actions*, if there exists a corresponding sequential execution $\langle \mathcal{A}, \Rightarrow \rangle$, such that for all i, if $\mathsf{count}(A_i) > k$ then $(s_i, A_i, s_{i+1}) \in \mathcal{S}$.

Note that the definition allows the first k actions of a processor to behave arbitrarily (even so far as to allow e.g. a read action to behave as a write action or vice versa), but the effect of such an arbitrary action should be globally consistent. Thus this definition gives the strong guarantee that all actions following such arbitrarily behaving actions will agree on how that action actually behaved. In particular, for $k = 0$ and in the absence of pending actions, the definition implies that all actions agree on the effect of the transient error on the state of the object; for example, for a shared register all reads that occur immediately after a transient error should return the same value.

Definition 5. An implementation of a shared object with sequential specification $\mathcal{S}$ is *k-stabilizing wait-free* if all its runs are wait-free and linearizable w.r.t. $\mathcal{S}$ after k processor actions.

In the above definition the stabilization delay k is taken to be independent of the type of operations performed by a processor, while one might very well feel that the difficulty of stabilizing different types of operations on the same object may vary. Indeed, preliminary versions of this definition were more fine-grained and included separate delays for different types of operations (e.g. allowing the first k_w writes and the first k_r reads performed by a processor on a read/write register to be arbitrary). It turns out that this amount of detail is really unnecessary, essentially because different types of operations on a shared object already need to reach some form of agreement on the state of the object.

The above definition is general and considers all objects whose behavior is described by a sequential specification. Since our goal is to study, from the lowest level, the requirements needed to support shared objects that are both wait-free and self-stabilizing, we need to define safe and regular self-stabilizing registers. A register is a shared object on which read operations R and write operations $W(v)$ are defined. For a run $\langle \mathcal{A}, \rightarrow \rangle$ over such a register, define the set $\mathcal{R}$ of read actions that do not act as writes, and define the set $\mathcal{W}$ of write actions plus read actions that act as writes. For $R \in \mathcal{R}$ define $\mathsf{val}(R)$ as the value returned by read action R and for $W \in \mathcal{W}$ define $\mathsf{val}(W)$ as the value 'actually' written by action W. Also for $W \in \mathcal{W}$ and $R \in \mathcal{R}$ define W *directly precedes* R, $W \rightleftarrows R$, if $W \rightarrow R$ and if there is no $W' \in \mathcal{W}$ such that $W \rightarrow W' \rightarrow R$. If no such write exists, we take the imaginary initial write $W_\perp$ responsible for writing the arbitrary initial value $\mathsf{val}(W_\perp)$. Let us write $A \| B$ if neither $A \rightarrow B$ nor $B \rightarrow A$. Define the *feasible* writes of a read R as all $W \in \mathcal{W}$ such that $W \rightleftarrows R$ or $W \| R$.

A write $W(v)$ on a register behaves correctly if $\mathsf{val}(W(v)) = v$. A read R on a *safe* register behaves correctly if $R \in \mathcal{R}$ and there is a write W such that $W \| R$, or $\mathsf{val}(R) = \mathsf{val}(W)$ for a write W with $W \rightleftarrows R$. A read on a *regular* register behaves correctly if $R \in \mathcal{R}$ and $\mathsf{val}(R) = \mathsf{val}(W)$ for some feasible write of R.

Definition 6. A safe or regular register is *k-stabilizing wait-free* if all its runs are wait-free and for all its runs only actions A with $\mathsf{count}(A) \leq k$ behave arbitrarily. Such a register is simply *stabilizing wait-free* if all its runs are wait-free and for all its runs only pending write actions W and read actions R overlapping pending writes behave arbitrarily *without* behaving as a write (i.e. $R \in \mathcal{R}$).

3 Stabilizing $1W1R$ Safe Bits Are Not Strong Enough.

In this section we prove that $1W1R$ safe bits are not strong enough to give self-stabilizing wait-free shared registers; this is shown by proving that there exists no implementation of a $1W1R$ wait-free k-stabilizing regular binary register using stabilizing $1W1R$ binary safe sub-registers.

It is a common convention to view the scheduling of processor steps as being chosen by an *adversary*, who seeks to force the protocol to behave incorrectly. The adversary is in control of (i) choosing the configuration of the system after a transient error and (ii) scheduling the processes' steps in a run.

The heart of the problem of such an implementation can be described as follows. Since the writer (the reader) cannot read the sub-registers which it can write, in order to know their contents and converge into correct stabilized behaviors, it has to rely on information that either is local or is passed to it through shared sub-registers that can be written only by the reader (the writer, respectively). The adversary can set the system in a state in which this information is inconsistent; subsequently, by scheduling the processes' sub-actions on the same sub-register to be concurrent, it can destroy the information propagation because of the weak consistency that safeness guarantees.

If an implementation of a $1W1R$ binary regular register from stabilizing $1W1R$ safe binary sub-registers exists, it must use two sets of sub-registers (that can be considered as two "big" sub-registers): one (S_W) that can be written by the writer and read by the reader and one (S_R) that can be written by the reader and read by the writer. A system configuration C is a tuple (L_R, L_W, S_R, S_W) that describes a system state, where L_R, L_W denote the reader's and writer's local states, respectively. Since there is a single reader, we assume that the value of the register depends only on the state of its implementation, i.e. for any configuration, the value of the register in that configuration is the value that would be returned by the $k+1$-th read of a sequence of reads taking place in a time interval during which no write action is executed. Similarly, we assume that the behaviour of an operation solely depends on the state in which it is executed and not in its position in the run, i.e. if a read behaves as a write in some state, it must behave as a write in that state wherever that state occurs in the run.

A read action on the regular register may involve several sub-reads of S_W; however, in the course for a contradiction, attention may be restricted to runs

in which all those sub-reads observe the same value of S_W, say sw. Then, we can consider that the value returned by each read is determined by a *reader's function* $f_R(lr, sw)$; let also $f_R^x(lr, sw)$ denote the value returned by the x-th read of a sequence of reads that start from a configuration where $L_R = lr$ and all find $S_W = sw$.

Theorem 7. *There exists no deterministic implementation of a wait-free k-stabilizing $1W1R$ binary regular register using $1W1R$ binary stabilizing safe sub-registers.*

Proof. Suppose that such an implementation exists. Since we look for a contradiction we may safely restrict attention to runs with no pending actions.

Consider an arbitrary initial configuration C and a run starting from C, in which the following actions are sequentially scheduled: k reads, k writes (writing arbitrary values), a $Write(0)$ action W_0 and a $Write(1)$ action W_1. The system configuration after W_0 is $C_0 = (lr, lw_0, sr, sw_0)$. Since the register is k-stabilizing by assumption the last of $k+1$ reads starting in C_0 must return 0, so $f_R^{k+1}(lr, sw_0) = 0$. Similarly, the last of $k+1$ reads starting after W_1 must return 1, Therefore, during the W_1 action the writer performs some sub-writes $s_1, \ldots, s_m$ on bits of S_W in that order. We write $sw \setminus s_1..s_i$ to denote the value of S_W after sub-writes $s_1, \ldots, s_i$ have been applied, while S_W held sw initially. Then $f_R^{k+1}(lr, sw_0 \setminus s_1..s_m) = 1$. Hence there will be an s_i $(1 \leq i \leq m)$, such that

$$f_R^{k+1}(lr, sw_0' = sw_0 \setminus s_1..s_{i-1}) = 0 \text{ and } f_R^{k+1}(lr, sw_1 = sw_0 \setminus s_1..s_i) = 1 \quad (1)$$

The former value for S_W could be observed by reads scheduled after s_{i-1} and before any other sub-operation of W_1, the latter by reads scheduled after s_i and before any other sub-operation of W_1. If $k+1$ reads are scheduled to take place overlapping s_i and they all observe sw_0' they will return 0 (Eq. 1); moreover, since the register is k-stabilizing by assumption, none of those reads should perform as write. Now let lr' be the local state of the reader after those reads and sw_1' the contents of S_W after completion of W_1 in that schedule. Again because the register is k-stabilizing,

$$f_R^{k+1}(lr', sw_1') = 1 \quad (2)$$

Now let the adversary set the system in $C_{01} = (lr, lw_0, sr, sw_1)$, i.e., differing from C_0 only in the contents of S_W, and schedule again a $Write(1)$ action. By our assumption and Eq. 1, the value of the register in C_{01} equals 1, while the writer observes the same state as before and, hence, it again performs the same sequence of subwrites. The adversary can again schedule $k+1$ reads overlapping s_i, as before; although the value of the corresponding bit does not change now during this subwrite, all these $k+1$ reads may observe sw_0' instead of sw_1', because of the safeness of the corresponding bit. From the previous paragraph and our initial assumption it is known that none of these reads performs as a write, while the $k+1$-th read returns 0 (Eq. 1). But the only feasible writes are a write of 1 (recall Eq. 2) and the write of the initial value, which is 1. This is a contradiction, as the $k+1$-th read must return the value of a feasible write.

S: stabilizing $1W2R$ **safe** bit

operation $Read() : \{0,1\}$
    return $(Read(S))$;

operation $Write(v : \{0,1\})$
$l : \{0,1\}$
    $l := Read(S)$;
    if $l \neq v$ **then** $Write(S, v)$;

Protocol 1. A stabilizing $1W1R$ regular bit

4 Self-Stabilizing Constructions of Shared Registers

The impossibility result of the previous section is based on the fact that the processes cannot read their own shared variables and that any information exchanged between them can be destroyed by the weakness of the safeness property. However, if we assume the existence of stabilizing dual-reader single-writer safe bits, this reasoning does not apply: to know the value of its own shared bits the writer can simply read them. This assumption is legitimate, because assuming a $1W2R$ safe bit exists is not much stronger than assuming a $1W1R$ safe bit exists. After all, the latter models a flip-flop with a single output wire, whereas the first models a flip-flop with its output wire split in two. We will formally prove in the next sections that if these $1W2R$ safe bits are used as basic building blocks in some well-known wait-free constructions of shared registers, the resulting constructions become, after minor modifications, self-stabilizing.

4.1 A Stabilizing $1W1R$ Regular Bit

Protocol 1 presents the adaptation of Lamport's [Lam86] construction of a $1W1R$ regular bit from a $1W1R$ safe bit, into a stabilizing wait-free one using a wait-free stabilizing $1W2R$ safe bit. We proceed by proving its correctness.

Theorem 8. *Protocol 1 implements a wait-free stabilizing $1W1R$ regular binary register using one wait-free stabilizing $1W2R$ safe binary register.*

Proof. Let $\langle \mathcal{A}, \rightarrow \rangle$ be an arbitrary run of reads R and writes W over the regular bit. Write $R(S)$ $(W(S))$ for the read from (write to) the safe bit S performed by read R (write W). Let $w_\perp$ be the initializing write of S, and set $\mathsf{val}(W_\perp) = \mathsf{val}(w_\perp)$ for the initializing write of the regular bit. If $\langle \mathcal{A}, \rightarrow \rangle$ has a pending write W, set $\mathsf{val}(W)$ to the value of S just after W; this is the value an interference free read starting after W will read. For all other, non-pending, writes set $\mathsf{val}(W(v)) = v$. According to Def. 6 it remains to show that in $\langle \mathcal{A}, \rightarrow \rangle$ all reads R not overlapping a pending write return the value written by a feasible write.

Claim 9. *If R does not overlap a pending write and $R(S)$ is interference free and $W \rightleftarrows R(S)$ then $\mathsf{val}(W) = \mathsf{val}(R(S))$.*

Proof. Let $W \rightleftarrows R(S)$ and let $\mathsf{val}(R(S)) = a$. If $W = W_\perp$ or W is a pending write, then $\mathsf{val}(W) = a$ by definition. Otherwise, note that a write $W(x)$ reads

$S_0 \ldots S_{l-1}$: stabilizing
    $1W1R$ **regular** bit

operation $Write(v : \{0, \ldots, l-1\})$
    $Write(S_v, 1)$;
    while $v \neq 0$
    do $v := v - 1$; $Write(S_v, 0)$;

operation $Read() : \{0, \ldots, l-1\}$
$w : \{0, \ldots, l\}$
    $w := 0$;
    while $Read(S_w) = 0 \wedge w < l$
    do $w := w + 1$;
    if $w = l$ **then return** $(l-1)$;
        else return (w) ;

Protocol 2. A stabilizing $1W1R$ l-ary regular register

S by $R'(S)$ without interference. Either $\mathsf{val}(R'(S)) = x$ so W does not write S or $\mathsf{val}(R'(S)) = \neg x$ and W does write x to S by $W(S)$ In the first case, as S is stabilizing and $R'(S)$ and $R(S)$ are not pending, and do not overlap a pending write, and there cannot be another write to S between $R'(S)$ and $R(S)$, $\mathsf{val}(R(S)) = \mathsf{val}(R'(S)) = x = \mathsf{val}(W)$. In the second case, as S is stabilizing and $W(S)$ not pending, and $R(S)$ is not pending and does not overlap a pending write, and $W(S) \rightleftarrows R(S)$, $\mathsf{val}(R(S)) = \mathsf{val}(W(S)) = x = \mathsf{val}(W)$.

Consider a read R not overlapping a pending write. Then $\mathsf{val}(R) = \mathsf{val}(R(S))$. If $R(S)$ is interference-free, then by Claim 9, for a write W with $W \rightleftarrows R(S)$ we have $\mathsf{val}(W) = \mathsf{val}(R(S)) = \mathsf{val}(R)$ and W is feasible for R. If $R(S)$ is interfered, there is a write $W(x)$ with $W \| R$ writing S and W cannot be pending by assumption. Then W read S by $R'(S)$ and $\mathsf{val}(R'(S)) = \neg x$. By Claim 9 and the fact that now $R'(S)$ is executed by a write so it cannot overlap another write, for W' with $W' \rightleftarrows W$ we must have $\mathsf{val}(W') = \neg x$. As $W \| R$, then $W' \rightleftarrows R$ or $W' \| R$, so both W and W' are feasible for R and $\mathsf{val}(R)$ equals one of these.

4.2 A Stabilizing $1W1R$ l-ary Regular Register

Protocol 2 presents the adaptation of Lamport's [Lam86] construction of a $1W1R$ l-ary regular register from l $1W1R$ regular bits, into a wait-free stabilizing one using l $1W1R$ wait-free stabilizing regular bits. We prove its correctness below.

Let $\langle \mathcal{A}, \rightarrow \rangle$ be an arbitrary run over the regular l-ary register. Number the writes consecutively, writing W^i for the write with index i. Let W^0 be the pending write if it exists, and $W_\perp$ otherwise. Let us write $R(S_v)$ for the read of S_v by read R, and let us write $W(S_v)$ for the write to S_v by a write W. The index of $W(S_v)$ equals the index of W (and the index of $w_{v,\perp}$ always equals 0). For reads R not overlapping a pending write, define $\pi(R)$ to be the largest index i such that W^i is feasible for R and $\mathsf{val}(W^i) = \mathsf{val}(R)$.

Consider the values of all S_v just after W^0 (i.e. the value read by an non-interfered read starting after W^0). Set $\mathsf{val}(W^0)$ to the minimal v such that $S_v = 1$, setting $\mathsf{val}(W^0) = l - 1$ if no such v exists. For all other writes $W(v)$ set $\mathsf{val}(W(v)) = v$. We first prove the following claim:

$S_{11} \ldots S_{nn}$: stabilizing $1W1R$ **regular**: $\mathbb{N} \times \{1, \ldots, n\} \times \mathcal{V}$
(with fields *tag*, *id*, and *val*)

operation $Write_i(v : \mathcal{V})$
$max : \mathbb{N} \times \{1, \ldots, n\} \times \mathcal{V}$
  $max := \max_{1 \le j \le n} Read(S_{ji})$;
  for $j := 1$ **to** n
  do $Write(S_{ij}, \langle max.tag + 1, i, v \rangle)$;

operation $Read_i() : \mathcal{V}$
$max : \mathbb{N} \times \{1, \ldots, n\} \times \mathcal{V}$
  $max := \max_{1 \le j \le n} Read(S_{ji})$;
  for $j := 1$ **to** n
  do $Write(S_{ij}, max)$;
  return $(max.val)$;

Protocol 3. A 1-stabilizing $nWnR$ l-ary atomic register

Claim 10. *Let R be a read not overlapping a pending write. If $W^i(u) \to R$ then $\pi(R(S_v)) \ge i$ for all $v \le u$. If R reads S_{w+1} then $\pi(R(S_w)) \le \pi(R(S_{w+1}))$.*

Proof. The first part easily follows as, for $i > 0$, $W^i(u)$ writes to all registers S_v with $v \le u$. For the second part, note that if R reads S_{w+1}, then $\mathsf{val}(R(S_w)) = 0$. If $\pi(R(S_w)) = i > 0$ (if $i = 0$ we are done immediately) then $\mathsf{val}(W^i(S_w)) = 0$ and hence W^i must write to S_{w+1} as well. Now by $W^i(S_w) \not\leftarrow R(S_w)$ and $W^i(S_{w+1}) \to W^i(S_w)$ and $R(S_w) \to R(S_{w+1})$ we get $W^i(S_{w+1}) \to R(S_{w+1})$ and hence $\pi(R(S_w)) = i \le \pi(R(S_{w+1}))$.

Theorem 11. *Protocol 2 implements a stabilizing $1W1R$ l-ary regular register using l stabilizing $1W1R$ regular binary registers.*

Proof. According to Def. 6 we have to show that in $\langle \mathcal{A}, \to \rangle$ all reads not overlapping a pending write return the value written by a feasible write. First consider a read R with $\mathsf{val}(R(S_v)) = 1$ for some v. Let $\pi(R(S_v)) = i$. Then $\mathsf{val}(R) = v$, $\mathsf{val}(W^i) = v$ and $W^i \not\leftarrow R$. W^i is not a feasible write for R only if there exists a $W^j(w)$ such that $W^i \to W^j(w) \to R$ (and so $i < j$). If $w \ge v$, then by Claim 10 $\pi(R(S_v)) \ge j > i$, and if $w < v$, then using Claim 10 inductively $i < j \le \pi(R(S_w)) \le \pi(R(S_v))$. This contradicts the assumption that $\pi(R(S_v)) = i$. Now consider a read R where, for all v, $\mathsf{val}(R(S_v)) = 0$. Then $\mathsf{val}(R) = l - 1$. Because all writes write 1, if anything, to S_{l-1}, and $\mathsf{val}(R(S_{l-1})) = 0$, we have $\pi(R(S_{l-1})) = 0$. Then, using Claim 10 inductively, $\pi(R(S_v)) = 0$ for all v, and so, for all v, the value of S_v just after W^0 equals 0. Hence, $\mathsf{val}(W^0) = l - 1$ by definition. By a similar argument as before, W^0 is feasible for R.

4.3 A 1-Stabilizing $nWnR$ l-ary Atomic Register

Protocol 3 presents the adaptation of the Vitányi-Awerbuch [VA86, AKKV88] multi-reader multi-writer atomic register construction from $1W1R$ multi-valued regular registers, into a wait-free 1-stabilizing $nWnR$ l-ary atomic register using n^2 wait-free stabilizing $1W1R$ ∞-ary regular registers. We prove its correctness below. In the protocol, $\mathcal{V}$ is the domain of values written and read by the multi-writer register. The construction uses n^2 regular stabilizing regular registers

S_{ij} written by processor i and read by processor j. These registers store a *label* consisting of an unbounded *tag*, a processor *id* with values in the domain $\{1, \ldots, n\}$, and a *value* in $\mathcal{V}$. Labels are lexicographically ordered by $\leq$.

The sequential specification of an atomic register simply states that a write updates the state to be the value written, whereas a read returns the state of the register. Let $\langle \mathcal{A}, \rightarrow \rangle$ be a run of the above protocol. In the remainder of the proof, $\langle \mathcal{A}, \rightarrow \rangle$ is the above run with actions A with $\mathsf{count}(A) = 0$ (thus $\mathsf{count}(A) \neq \oplus$) removed. We will show that for this "sub-run" there exists a corresponding sequential execution $\langle \mathcal{A}, \Rightarrow \rangle$ such that $\Rightarrow$ is an extension of $\rightarrow$ and for all reads R with $\mathsf{count}(R) > 1$, R returns the current state of the register. As for actions A, B with $\mathsf{count}(A) = 0$ and $\mathsf{count}(B) \neq 0$ in the original run either $A \rightarrow B$ or $A \| B$, we can prepend all these A to $\langle \mathcal{A}, \Rightarrow \rangle$ such that the resulting sequential execution $\langle \mathcal{A}, \Rightarrow \rangle$ corresponds to the original run $\langle \mathcal{A}, \rightarrow \rangle$ and satisfies Def. 4.

We are going to partition $\mathcal{A}$ into a set $\mathcal{R}$ of actions that behave as reads and a set $\mathcal{W}$ of actions that behave as writes. To this end, define

$$\begin{aligned} \mathcal{F} &= \{A \in \mathcal{A} \mid \mathsf{count}(A) = \oplus \ \vee \ \mathsf{count}(A) = 1\} \\ \mathcal{R}^- &= \{A \in \mathcal{A} \mid \mathsf{count}(A) > 1 \text{ and } A \text{ is a read}\} \\ \mathcal{W}^- &= \{A \in \mathcal{A} \mid \mathsf{count}(A) > 1 \text{ and } A \text{ is a write}\} \end{aligned}$$

Then $\mathcal{F}$ corresponds to the set of actions that, according to Def. 4, may behave arbitrary. We further subdivide $\mathcal{F}$ into actions $\mathcal{F}_W$ that seem to behave as a write and actions $\mathcal{F}_R$ that seem to behave as a read, making sure that no two apparent writes write the same label (because the remainder of the proof, especially the definition of the reading function π depends on this). Define for $A \in \mathcal{A}$, $\mathsf{label}(A)$ as the label written by A, and for a set of actions $\mathcal{F}$, $\mathsf{label}(\mathcal{F}) = \{\mathsf{label}(F) \mid F \in \mathcal{F}\}$. Set $\mathcal{L} = \mathsf{label}(\mathcal{F}) \setminus \mathsf{label}(\mathcal{W}^-)$ and let $\mathcal{F}_W$ be an arbitrary subset of $\mathcal{F}$ such that

$(F1)$ $\mathsf{label}(\mathcal{F}_W) = \mathcal{L}$, and
$(F2)$ For all $A, B \in \mathcal{F}_W$, if $\mathsf{label}(A) = \mathsf{label}(B)$ then $A = B$, and
$(F3)$ For all $A \in \mathcal{F}_W$ and $B \in \mathcal{F}$, if $\mathsf{label}(A) = \mathsf{label}(B)$ then $t_i(A) < t_i(B)$.

Now set $\mathcal{F}_R = \mathcal{F} \setminus \mathcal{F}_W$ and define $\mathcal{W} = \mathcal{W}^- \cup \mathcal{F}_W$ and $\mathcal{R} = \mathcal{R}^- \cup \mathcal{F}_R$.

Lemma 12. *If $A \rightarrow B$ then $\mathsf{label}(A) \leq \mathsf{label}(B)$. If $B \in \mathcal{W}^-$ this inequality is strict.*

Proof. Let A be performed by processor i and B be performed by processor j. If $A \rightarrow B$, then the write to S_{ij} by A precedes the read of S_{ij} by B. Because we only consider actions with $\mathsf{count} \neq 0$, the write to S_{ij} is not pending, and by $A \rightarrow B$ the read of S_{ij} does not overlap a pending write. Then the write of A to S_{ij} or a later write by action C of i to S_{ij} is a feasible write to the read of S_{ij} of B—hence this read returns the value written to S_{ij} by processor i during action A or the later action C. Since processor i both reads and writes from S_{ii}, and $\mathsf{count}(A) \neq 0$, $\mathsf{label}(A) \leq \mathsf{label}(C)$. Therefore the read of S_{ij} by B returns a label greater than or equal to $\mathsf{label}(A)$. B picks the maximum of all labels read, so if B is a read, $\mathsf{label}(A) \leq \mathsf{label}(B)$ and if B is a write, then $\mathsf{label}(A) < \mathsf{label}(B)$.

Lemma 13. *For all $R \in \mathcal{R}$ there exists a $W \in \mathcal{W}$ such that* $\mathsf{label}(W) = \mathsf{label}(R)$ *and $R \not\rightarrow W$.*

Proof. Define $A \rightsquigarrow B$ iff $\mathsf{label}(A) = \mathsf{label}(B)$ and $B \not\rightarrow A$. Then $A \rightsquigarrow A$. Let $R \in \mathcal{R}$ be arbitrary, and pick a $B \in \mathcal{A}$ such that $B \rightsquigarrow R$ and for no $A \in \mathcal{A}$, $A \neq B$, $A \rightsquigarrow B$. If $B \in \mathcal{W}$ we are done, so assume $B \in \mathcal{R}$. Suppose $\mathsf{count}(B) > 1$. Then there is an operation C on the same processor with $\mathsf{count}(C) = 1$ and $C \rightarrow B$. If $\mathsf{label}(C) = \mathsf{label}(B)$ then $C \rightsquigarrow B$, while if $\mathsf{label}(C) < \mathsf{label}(B)$ (the only other possible case according to Lemma 12) then the contents of the register from which B obtains $\mathsf{label}(B)$ has changed after C read that same register. This register then is written by an operation D with $\mathsf{label}(D) = \mathsf{label}(B)$ before B reads it. Then $D \not\rightarrow C$, which, as $\mathsf{count}(C) = 1$, implies $\mathsf{count}(D) \neq 0$ and hence $D \rightsquigarrow B$. This contradicts the assumption that there is no A such that $A \rightsquigarrow B$.

We conclude that $\mathsf{count}(B) \leq 1$ and hence $B \in \mathcal{F}$, so $\mathsf{label}(B) \in \mathsf{label}(\mathcal{F})$. So either there exists a $W \in \mathcal{W}^-$ such that $\mathsf{label}(W) = \mathsf{label}(B) = \mathsf{label}(R)$, or $\mathsf{label}(B) \in \mathcal{L}$ and by $(F1)$ there exists a $W' \in \mathcal{F}_W$ with $\mathsf{label}(W') = \mathsf{label}(B) = \mathsf{label}(R)$. In the first case, by Lemma 12, $R \not\rightarrow W$ as required. In the second case, since $B \in \mathcal{F}$ we must have by $(F3)$, $t_i(W') < t_i(B)$. Then as $B \rightsquigarrow R$ implies $R \not\rightarrow B$, this in turn implies $R \not\rightarrow W'$.

Lemma 14. *For all $W, W' \in \mathcal{W}$ if* $\mathsf{label}(W) = \mathsf{label}(W')$ *then $W = W'$.*

Proof. There are three cases

$W, W' \in \mathcal{W}^-$**:** By the protocol then W and W' must be executed by the same processor (or else their *id*-fields differ). But then either $W \rightarrow W'$ or $W' \rightarrow W$. By Lemma 12 then $\mathsf{label}(W) \neq \mathsf{label}(W')$, a contradiction.

$W \in W^-, W' \in \mathcal{F}_W$**:** If $W \in W^-$ then $\mathsf{label}(W) \notin \mathcal{L}$, and if $W' \in \mathcal{F}_W$ then $\mathsf{label}(W') \in \mathcal{L}$ by $(F1)$. Therefore $\mathsf{label}(W) \neq \mathsf{label}(W')$, a contradiction.

$W, W' \in \mathcal{F}_W$**:** If $\mathsf{label}(W) = \mathsf{label}(W')$, then by $(F2)$ we have $W = W'$.

Now we can define a reading mapping $\pi : \mathcal{R} \mapsto \mathcal{W}$ for a particular run $\langle \mathcal{A}, \rightarrow \rangle$ by $\pi(R) = W$ if $\mathsf{label}(R) = \mathsf{label}(W)$ and $W \in \mathcal{W}$.

Lemma 15. *For all $R \in \mathcal{R}$, $\pi(R)$ is defined and unique, $R \not\rightarrow \pi(R)$, and R returns the value written by $\pi(R)$.*

Proof. That $\pi(R)$ is defined and $R \not\rightarrow \pi(R)$ follows from Lemma 13. That it is unique follows from Lemma 14. If $\pi(R) \in \mathcal{W}^-$, then $\mathsf{label}(\pi(R)).val$ equals the value written by $\pi(R)$. If $\pi(R) \in \mathcal{F}_W$ we define the (arbitrary) value written by $\pi(R)$ to equal $\mathsf{label}(\pi(R)).val$

We now show that every run $\langle \mathcal{A}, \rightarrow \rangle$ with the above reading function π is atomic. Define for $W \in \mathcal{W}$ its clan $[W]$ by $[W] = \{W\} \cup \{R \in \mathcal{R} \mid \pi(R) = W\}$, and let $\Gamma = \{[W] \mid W \in \mathcal{W}\}$ be the set of all clans. Define $\rightarrow'$ over Γ by

$$[W] \rightarrow' [W'] \iff (\exists A \in [W], B \in [W'] :: A \rightarrow B)$$

Lemma 16. *For all $W \in \mathcal{W}$ and $A, B \in [W]$ we have* $\mathsf{label}(A) = \mathsf{label}(B)$. *Also if $W \neq W'$, then for all $A \in [W], B \in [W']$ we have* $\mathsf{label}(A) \neq \mathsf{label}(B)$.

Proof. The first part follows from the definition of $[W]$ and $\pi(R)$. The second part follows from Lemma 14.

Lemma 17. *$\rightarrow'$ is an acyclic partial order over Γ.*

Proof. Suppose not. Then there exists a chain

$$[W_1] \rightarrow' [W_2] \rightarrow' \cdots \rightarrow' [W_m] \rightarrow' [W_1]$$

with $m > 1$, and $W_i \neq W_j$ if $i \neq j$. This implies that for all i with $1 \leq i \leq m$ there exist actions $A_i, B_i \in [W_i]$ such that $A_i \rightarrow B_{i+1}$ (addition modulo $m+1$ from now). By Lemma 12 and 16 $\mathsf{label}(A_i) \leq \mathsf{label}(B_{i+1}) = \mathsf{label}(A_{i+1})$. Then $\mathsf{label}(A_1) = \mathsf{label}(A_2)$, contrary to Lemma 16.

Applying these lemmas and the results of [AKKV88] we get the result.

Theorem 18. *Protocol 3 implements a 1-stabilizing $nWnR$ l-ary atomic register using n^2 stabilizing $1W1R$ ∞-ary regular registers.*

Proof. Define a total order $\Rightarrow$ over $\mathcal{A}$ extending $\rightarrow$ as follows. First extend $\rightarrow'$ over Γ to a total order $\Rightarrow'$ (according to Lemma 17, this is possible). Now for $A \in [W]$ and $B \in [W']$ let $A \Rightarrow B$ if $[W] \Rightarrow' [W']$ (a). This extends $\rightarrow$ because if $A \rightarrow B$, then by the definition of $\rightarrow'$, $[W] \rightarrow' [W']$ and thus $[W] \Rightarrow' [W']$. For $A, B \in [W]$ fix an arbitrary extension $\Rightarrow$ of $\rightarrow$ such that for the only writer $W \in [W]$ we have for all other $C \in [W]$ that $W \Rightarrow C$ (b). This is an extension of $\rightarrow$ because by Lemma 15, $C \not\rightarrow W$. Now $\Rightarrow$ is a total order over $\mathcal{A}$ such that for all $R \in \mathcal{R}$ $\pi(R) \Rightarrow R$ by Lemma 15 and (b). Also there does not exists a $W \in \mathcal{W}$ such that $\pi(R) \Rightarrow W \Rightarrow R$, because by (a) and the fact that $R \notin [W]$ by Lemma 16, either $W \Rightarrow [\pi(R)]$ or $R \Rightarrow [W]$. Hence $W \Rightarrow \pi(R)$ or $R \Rightarrow W$.

5 Further Research

Our results are a first step towards exploring the relation between self-stabilization and wait-freedom in the construction of shared objects. There are still a lot of interesting questions in this new area that remain unanswered. First of all, this paper describes a first attempt to propose a reasonable and general definition of a self-stabilizing wait-free shared object. Although we believe our approach is viable, further research is necessary to demonstrate this, or to decide that other definitions may be more appropriate. Second, our construction of the 1-stabilizing $nWnR$ atomic register uses unbounded time-stamps to invalidate old values. We would like to know whether this necessarily so, or if the space requirements of a k-stabilizing atomic register can be bounded. Finally, following the work of Aspnes and Herlihy [AH90], it is an interesting venture to classify, based on their sequential specification, all k-stabilizing shared memory objects that can be constructed from k'-stabilizing atomic registers, and to provide a general method to do so.

Acknowledgements

It's a pleasure to thank Moti Yung for his encouragement in this work. We are grateful to the anonymous referees for their accurate and insightful comments, and to the MPI and the CWI for their hospitality during mutual visits.

References

[AGMT92] Afek, Y., Greenberg, D., Merritt, M., and Taubenfeld, G. Computing with faulty shared memory. In *11th PODC* (Vancouver, BC, Canada, 1992), ACM Press, pp. 47–58.

[AKY90] Afek, Y., Kutten, S., and Yung, M. Memory-efficient self stabilizing protocols for general graphs. In *4th WDAG* (Bari, Italy, 1990), LNCS 486, Springer Verlag, pp. 15–28.

[AMT93] Afek, Y., Merritt, M., and Taubenfeld, G. Benign failure models for shared memory. In *7th WDAG* (Lausanne, Switzerland, 1993), LNCS 725, Springer Verlag, pp. 69–83.

[AH93] Anagnostou, E., and Hadzilacos, V. Tolerating transient and permanent failures. In *7th WDAG* (Lausanne, Switzerland, 1993), LNCS 725, Springer Verlag, pp. 174–188.

[AH90] Aspnes, J., and Herlihy, M. P. Wait-free data structures in the asynchronous PRAM model. In *2nd SPAA* (Crete, Greece, 1990), ACM Press, pp. 340–349.

[AKKV88] Awerbuch, B., Kirousis, L. M., Kranakis, E., and Vitányi, P. M. B. A proof technique for register atomicity. In *8th FST&TCS* (Pune, India, 1988), LNCS 338, Springer Verlag, pp. 286–303.

[AKM+93] Awerbuch, B., Kutten, S., Mansour, Y., Patt-Shamir, B., and Varghese, G. Time optimal self-stabilizing synchronization. In *25th STOC* (San Diego, CA, USA, 1993), ACM Press, pp. 652–661.

[BGW89] Brown, G. M., Gouda, M. G., and Wu, C. L. Token systems that self-stabilize. *IEEE Trans. on Comput.* **38**, 6 (1989), 845–852.

[BP89] Burns, J. E., and Pachl, J. Uniform self-stabilizing rings. *ACM Trans. Prog. Lang. & Syst.* **11**, 2 (1989), 330–344.

[Dij74] Dijkstra, E. W. Self-stabilizing systems in spite of distributed control. *Comm. ACM* **17**, 11 (1974), 643–644.

[DIM93] Dolev, S., Israeli, A., and Moran, S. Self-stabilization of dynamic systems assuming only read/write atomicity. *Distr. Comput.* **7**, 1 (1993), 3–16.

[DW93] Dolev, S., and Welch, J. L. Wait-free clock synchronization. In *12th PODC* (Ithaca, NY, USA, 1993), ACM Press, pp. 97–108.

[GP93] Gopal, A. S., and Perry, K. J. Unifying self-stabilization and fault-tolerance. In *12th PODC* (Ithaca, NY, USA, 1993), pp. 195–206.

[Her91] Herlihy, M. P. Wait-free synchronization. *ACM Trans. Prog. Lang. & Syst.* **13**, 1 (1991), 124–149.

[Hoe94] Hoepman, J.-H. Uniform deterministic self-stabilizing ring-orientation on odd-length rings. In *8th WDAG* (Terschelling, The Netherlands, 1994), LNCS 857, Springer Verlag, pp. 265–279.

[IJ93] Israeli, A., and Jalfon, M. Uniform self-stabilizing ring orientation. *Inf. & Comput.* **104**, 2 (1993), 175–196.

[IS92] Israeli, A., and Shaham, A. Optimal multi-writer multi-reader atomic register. In *11th PODC* (Vancouver, BC, Canada, 1992), ACM Press, pp. 71–82.

[JCT92] Jayanti, P., Chandra, T., and Toueg, S. Fault-tolerant wait-free shared objects. In *33rd FOCS* (Pittsburgh, Penn., USA, 1992), IEEE Comp. Soc. Press, pp. 157–166.

[KP90] Katz, S., and Perry, K. J. Self-stabilizing extensions for message-passing systems. In *9th PODC* (Quebec City, Quebec, Canada, 1990), ACM, ACM Press, pp. 91–101.

[Lam86] Lamport, L. On interprocess communication. Part I: Basic formalism, part II: Algorithms. *Distr. Comput.* **1**, 2 (1986), 77–101.

[LTV89] Li, M., Tromp, J., and Vitányi, P. M. B. How to share concurrent wait-free variables. Tech. Rep. CS-R8916, CWI, Amsterdam, 1989.

[LV91] Li, M., and Vitányi, P. M. B. Optimality of wait-free atomic multiwriter variables. Tech. Rep. CS-R9128, CWI, Amsterdam, The Netherlands, 1991.

[PT94] Papatriantafilou, M., and Tsigas, P. Wait-free self-stabilizing clock synchronization. In *4th SWAT* (Århus, Denmark, 1994), LNCS 824, Springer Verlag, pp. 267–277.

[PB87] Peterson, G. L., and Burns, J. E. Concurrent reading while writing ii: The multi-writer case. In *28th FOCS* (Los Angeles, CA, USA, 1987), IEEE Comp. Soc. Press, pp. 383–392.

[Sch93] Schneider, M. Self-stabilization. *ACM Comput. Surv.* **25**, 1 (1993), 45–67.

[Tel94] Tel, G. *Introduction to Distributed Algorithms.* Cambridge University Press, 1994.

[VA86] Vitányi, P. M. B., and Awerbuch, B. Atomic shared register access by asynchronous hardware. In *27th FOCS* (Toronto, Ont., Canada, 1986), IEEE Comp. Soc. Press, pp. 233–243.

This article was processed using the LaTeX macro package with LLNCS style