Self-Stabilization of Wait-Free Shared Memory Objects*

Jaap-Henk Hoepman
Department of Computer Science
University of Twente, the Netherlands
hoepman@cs.utwente.nl

Marina Papatriantafilou
Department of Computing Science
Chalmers University of Technology, Sweden
ptrianta@cs.chalmers.se

Philippas Tsigas
Department of Computing Science
Chalmers University of Technology, Sweden
tsigas@cs.chalmers.se

March 2, 2001

Abstract

This paper proposes a general definition of self-stabilizing wait-free shared memory objects. The definition ensures that, even in the face of processor failures, every execution after a transient memory failure is linearisable except for a bounded number of actions.

Shared registers have been used extensively as communication medium in self-stabilizing protocols. We give particular attention to the self-stabilizing implementation of such registers, thus providing a large body of previous research with a more solid fundament.

In particular, we prove that one cannot construct a self-stabilizing single-reader single-writer regular bit from self-stabilizing single-reader single-writer safe bits, using only a single bit for the writer. This leads us to postulate a self-stabilizing *dual*-reader single-writer safe bit as the minimal hardware needed to achieve self-stabilizing wait-free interprocess communication and synchronisation. Based on this hardware, adaptations of well known wait-free implementations of regular and atomic shared registers are proven to be self-stabilizing.

1 Introduction

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

*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). A preliminary version of this paper appeared as [HPT95].

the first model a distributed system must remain operational while a certain fraction of the processors is malfunctioning. When constructing shared memory objects like 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 [AH90, Her91]. 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* [Dij74, Sch93].

To develop truly reliable systems both failure models must be considered together. Research in this area has started to emerge (see Section 1.2 for an overview). However, these works have taken the existence of a fault tolerant and self-stabilizing means of communication (either by message exchange or through shared memory) for granted. Also, most research on self-stabilization uses shared memory – and shared registers in particular – for communication, to simplify reasoning about the protocols under study. Again, the shared memory objects themselves are implicitly assumed to be self-stabilizing. Moreover, no consideration is given to so called corrupted actions that are active while a transient error occurs. Instead it is assumed that all atomic actions happen instantaneously and that any such error occurs only *in between* calls to atomic actions.

It is not immediately clear that these assumptions are indeed reasonable. Any operation, even atomic ones, do take some time to complete, during which an error can occur. This paper, however, shows that these assumptions are valid by thoroughly exploring the relation between self-stabilization and wait-freedom in shared memory objects, especially shared registers.

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) z-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.

1.1 Summary of results

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. Our work, finally, provides a solid fundament for the use of shared registers as the basis of communication in most previous research on self-stabilization. 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 hardware needed for a system to be able to converge to legal behaviours after transient memory faults, as well as to remain operative in the presence of processor crashes. Moreover, registers are extensively used in the construction of even more complex shared memory objects, and are the primary means of communication in most self-stabilizing protocols.

Our contribution in this paper is fourfold. First and foremost, 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. This definition is not limited to shared registers only, but encompasses all possible shared memory objects that can be defined by a sequential specification. 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, if we restrict the writer to write only a single bit. We conjecture that this is also impossible if we allow an arbitrary number of bits written by the writer. Note that single-reader single-writer safe bits have traditionally been used as the basic building blocks in wait-free shared register implementations.

This leads us to postulate a self-stabilizing *dual*-reader single-writer safe bit, which, from a hardware point of view, resembles a flip-flop with its output wire split in two (cf. Sect. 4). Moreover, in the same section we prove that no construction of multi-reader registers from single-reader registers can be made to stabilize immediately after a transient error. The first operation of each processor must be allowed to behave arbitrarily. Finally, using the dual-reader safe bit as a basic building block, we formally prove 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 a thorough discussion of our results and directions for further research.

1.2 Related work

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-synchronisation protocols is presented in [DW93, PT94].

Another approach for combining processor and memory failures is put forward by Afek *et al.* [AGMT92, AMT93] and Jayanti *et al.* [JCT92]. They analyse 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 on the 'high-level' object during which more than t 'low-level' objects fail are required to fail in the same manner as these 'low-level' objects.

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. We feel that this notion of a self-stabilizing object does not agree well with the additional requirement that the object is wait-free. On wait-free shared objects, a single processor can make progress even if all other processors have crashed. This definition of self-stabilization, on the other hand, only guarantees recovery from transient errors in fair runs (in which no processors crash). Moreover, both Li and Vitányi [LV91] and Israeli and Shaham [IS92] do not consider the possibility of corrupted operations (that are already active when the transient error occurs).

2 Defining self-stabilizing wait-free objects

In the definition of shared memory objects we follow the concept of *linearizability* (cf. [HW90, Her91]), which we, for the sake of self containment, briefly paraphrase here. We will then discuss our processor model and how this affects *implementing* shared memory objects in Section 2.1, and proceed with our new definition of self-stabilizing shared memory objects in Section 2.2. Finally, we will discuss our model in Section 2.4.

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 query 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 v on its invocation and returns a value r as its response ($r = O(v)$). We sometimes write O_p to indicate that processor p is executing operation O[1]. Each such operation execution is called an *action*, and takes a non-zero amount of time to complete. 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). For incomplete actions, $t_R(A) = \infty$ and the response is unknown. An object is *wait-free* if an action always can complete within an a priori bounded amount of time, irrespective of the actions or crashes of other processor.

The desired behaviour of an object is described by its *sequential specification* $\mathcal{S}$. This specifies the set of possible states of the object, the initial state of the object, and for each operation its effect on the state and its (optional) response. We write $(s, r = O(v), s') \in \mathcal{S}$ if invoking O with parameters v 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 $t_R(A) < t_I(B)$. If two actions A, B are incomparable under $\rightarrow$, i.e., if neither $A \rightarrow B$ nor $B \rightarrow A$, they are said to *overlap*. Then we write $A \| B$. We write $A \not\leftarrow B$ if $A \rightarrow B$ or $A \| B$. Because processors are sequential, we require for every run that no two actions A and B executed by the same processor overlap. Furthermore, runs start with no processor accessing the object. Runs have infinite length, and capture the real time ordering — as could be observed externally without a stopwatch — between actions invoked by the processors.

[1]Although the notation might imply otherwise, we consider $O_p(u)$ and $O_p(w)$ to be *different* operations. In other words, both the processor p executing O and the actual parameters v to O are part of the name $O_p(v)$. The set $\mathcal{O}$ contains all these different operations.

A run is called *complete*, if all actions that start in it have finished in it, i.e., if no crashes occurred. A non-complete run can be completed by including an arbitrary response for each pending action, where the inserted responses are ordered after all other actions in the run, and ordered arbitrarily among each other[2]. This complete run is called the *completion* of the run. A run may have many completions (by using all possible values for the unknown responses).

A *sequential execution* $\langle \mathcal{A}, \Rightarrow \rangle$ over the object is a run where all actions are totally ordered according to the transitive irreflexive order $\Rightarrow$. In other words, $\mathcal{A}$ consists of actions $A_1, A_2, \ldots$ and $A_i \Rightarrow A_j$ if and only if $i < j$. A run $\langle \mathcal{A}, \rightarrow \rangle$ *corresponds* to 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. Note that there may be more than one sequential execution corresponding to a single run, because the order between two concurrent actions can be fixed either way. A sequential execution *satisfies* sequential specification $\mathcal{S}$ if there exists a sequence of states $s_1, s_2, \ldots$ such that s_1 is the initial state of $\mathcal{S}$ and (s_i, A_i, s_{i+1}) in $\mathcal{S}$ for all A_i in $\mathcal{A}$.

Definition 2.1 *A run $\langle \mathcal{A}, \rightarrow \rangle$ over an object is* linearisable w.r.t. sequential specification $\mathcal{S}$, *if for at least one of its completions there exists a corresponding sequential execution $\langle \mathcal{A}, \Rightarrow \rangle$ that satisfies $\mathcal{S}$.*

An object is linearisable w.r.t. its sequential specification $\mathcal{S}$ if all possible runs over the object are linearisable w.r.t. $\mathcal{S}$. Informally speaking, an object is linearizable w.r.t. to specification $\mathcal{S}$ if all actions appear to take effect instantaneously and act according to $\mathcal{S}$. The challenge in implementing such linearizable objects is to avoid all non-linearizable runs.

2.1 Implementing shared memory objects

When implementing compound objects from lower level ones, the operations of the compound object are implemented by means of a sequential procedure that can invoke an operation on one of the primitive objects or do some local computations. The implementation must be such that from the known properties of the primitive objects and the order of the steps taken by the procedure, correct behaviour of the compound object can be proven. Processors are sequential and, therefore, cannot invoke an action if their previously invoked action has not responded yet. An implementation of a shared object is *wait-free* if each invocation of an operation by a processor can respond within a bounded number of steps, irrespective of the behaviour of the other processors. This includes the case when all other processors have crashed.

A complication arises when we consider transient errors that can corrupt an arbitrary part of the system state, including, for instance, the programming counter or other parts of the internal CPU state of one or more of the processors. We limit the effect of such errors by viewing the abstract processor (or indeed an arbitrary shared memory object) to be an IO automaton [ALR88, Her91] in the following way.

[2] Pending actions may have had an effect on other concurrent actions, and therefore should be 'serialised' in time before these actions in the corresponding sequential execution. In order to verify that the sequential specification is satisfied, the arbitrary response of this action must be fixed.

A shared memory object is an IO automaton, that has a distinct input action for each possible combination of operation, invoking processsor, and possible parameter values. For each input action, one or more distinct output actions are defined that signal the end of the operation execution and the return value if defined. An implementation of a shared memory object is again an IO automaton that defines internal actions, states and a transition function such that when it is combined with the IO automata implementing the lower level objects, each input action results in an output action such that the resulting run is linearizable.

In this model, the adversary controlling the effect of the transient error can only pick the state of the whole IO automaton, and execution starts from this state. Translating this model to ordinary system architectures boils down to the following. Based on the current state, the program to execute and the current program counter, the instruction scheduler decides which instruction to execute next. Both the machine state and the current program counter can be changed by the adversary controlling the effect of the transient error. In particular, the adversary can pick an arbitary instruction and start executing this instruction, independent of the program state. However, the adversary cannot pick the current program counter independent of the currently executing instruction (thus further influencing the selection of the next instruction to schedule). Instead, the program counter is guaranteed always to correspond to the instruction currently executed. Implementations of such a scheduler could enforce this by verifying at the end of each instruction execution that the current program counter indeed points to the instruction executed last, using the uncorrupted program segment.

This prevents the adversary from executing an arbitrary instruction and then selecting a completely unrelated instruction as the start of the executing following the error. Instead, both instructions must be possible successors as described by the program executed. This restriction is exploited by two of our constructions, using a fixed programmed sequence of actions instead of while- or for-loops (cf. Section 4.2). Note also, that in this model calls to operations on lower level objects should be interpreted as *macros*, not subroutine calls, because otherwise the return address could be corrupted by the adversary.

2.2 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 2.2 *A shared wait-free object is self-stabilizing if every* fair *run (in which all operations on all processors are executed infinitely often) that is 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. This clearly violates the wait-free property which states 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 2.3 *A shared wait-free object is self-stabilizing, if every run 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 occur 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 *corrupted* actions A, and for such actions alone, we set $t_I(A) = 0$.

Consider a run $\langle \mathcal{A}, \rightarrow \rangle$. For all $A \in \mathcal{A}$, define $\mathsf{count}(A) = 0$ for all corrupted actions A, i.e. with $t_I(A) = 0$, and define $\mathsf{count}(A)$ equal to i if A is executed by p as its i-th non-corrupted action. Then the first action A executed by processor p either has $\mathsf{count}(A) = 0$ (if it is corrupted) or $\mathsf{count}(A) = 1$ (if not). As each processor executes sequentially, and actions are unique and executed by a single processor[3], count is well-defined.

We are now ready to present the formal definition of a self-stabilizing wait-free shared memory object.

Definition 2.4 *A run $\langle \mathcal{A}, \rightarrow \rangle$ is* linearizable w.r.t. sequential specification $\mathcal{S}$ after k processor actions, *if for some completion of the run there exists a corresponding sequential execution $\langle \mathcal{A}, \Rightarrow \rangle$ and a sequence of states $s_1, s_2, \ldots$, such that for all $A_i \in \mathcal{A}$, if $\mathsf{count}(A_i) > k$ then $(s_i, A_i, s_{i+1}) \in \mathcal{S}$.*

Definition 2.5 *A shared object is* k-stabilizing wait-free *with sequential specification $\mathcal{S}$ if the object is wait-free and all its runs are linearizable w.r.t. $\mathcal{S}$ after k processor actions.*

We call k the *stabilization delay* of the object.

These definitions allow all corrupted actions, and the first k actions of each processor to behave arbitrarily (even so far as to allow e.g. a read action to behave as a write action or vice versa). However, the effect of such an arbitrary action should be globally consistent.

Observe also that in a sequential execution, the count values of the actions may not be monotonically increasing. Fast processors may reach the stabilization delay k of the object much earlier in the sequential execution than some slow processor. In this case the actions may behave according to the sequential specification for a long time (their count value being larger than k), until a slow processor executes an action A with $\mathsf{count}(A) \leq k$ changing the object state in an arbitrary way. Again, this change should be globally consistent, in the sense that all fast processors must agree on the effect of this action A.

We see that all fast processors must agree — after k operations — on the arbitrary behaviour of the slow processors. In particular, for $k = 0$ and in the absence of corrupted actions, the definition implies that all actions should reach agreement on the effect of the transient error on the state of the object. For example, for a 0-stabilizing shared register all reads that occur immediately after a transient error should return the same value. Intuitively, one could say that 0-stabilization means that each processor has at most one corrupted initial action.

Definition 2.5 is general and considers all *atomic* objects whose behaviour is described by a sequential specification. Because linearizability can be and generally is used to specify the behaviour of arbitrary wait-free atomic shared memory objects, our extension of these definitions to include self-stabilization is equally universal.

[3]There is no concept like joint actions as in CSP

Of course, we want to know whether such self-stabilizing shared objects exist and how they can be implemented. Traditionally, in the non self-stabilizing case, atomic objects have been built using several layers. Starting with safe registers as the minimal hardware available for interprocess communication, regular registers, multi-reader registers and finally atomic registers were constructed. We take a similar approach. Therefore, we also need to define when safe and regular registers are self-stabilizing, as their behaviour is not described by a sequential specification.

2.3 Shared registers: safeness and regularity

A register is a shared object on which read operations R and write operations $W(v)$ are defined. A nWmR register is a register that may be written by n processors and may be read by m processors. Except for 1W1R registers, we otherwise assume that a processor writing a register can also read this register using a read operation (and not by examining its local state, as is customary to assume). In particular, a 1W2R register is one that can be written by a single processor, and can be read by two processors, one of which is the writer.

For a run $\langle \mathcal{A}, \to \rangle$ over such a register, partition the set of actions $\mathcal{A}$ into a set of reads $\mathcal{R}$ and a set of writes $\mathcal{W}$, where $\mathcal{R}$ may contain 'bad' writes (behaving as reads) whereas $\mathcal{W}$ may contain 'bad' reads (behaving as writes). Let $\mathcal{V}$ be the value-domain of the register, and fix an arbitrary value assignment $\mathsf{val}(\cdot)$ from the set of actions $\mathcal{A}$ to $\mathcal{V}$. Also for $W \in \mathcal{W}$ and $R \in \mathcal{R}$ define W *directly precedes* R, $W \rightleftharpoons R$, if $W \to R$ and if there is no $W' \in \mathcal{W}$ such that $W \to W' \to R$. If no such write exists, we take the imaginary initial write $W_\perp$ responsible for writing the arbitrary initial value $\mathsf{val}(W_\perp)$. Define the *feasible* writes of a read R as all $A \in \mathcal{W}$ such that $A \rightleftharpoons R$ or $A \| R$.

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

Definition 2.6 *A safe or regular register is* k-stabilizing wait-free *if the register is wait-free and for all its runs the set of actions can be partitioned in sets $\mathcal{R}$ and $\mathcal{W}$ and can be assigned values $\mathsf{val}(\cdot)$ such that all actions A with $\mathsf{count}(A) > k$ behave correctly.*

2.4 Discussion

The definition of self-stabilizing wait-free shared memory objects given in the previous section is only one possible definition for such objects. We will later show that this definition is viable, by presenting constructions of shared registers from weaker ones that satisfy this definition. In this section we discuss possible variations on the definition, and explain our particular choice of definition.

First, we initially thought that the corrupted actions could pose serious problems to the self-stabilization of shared memory objects. Indeed, in [HPT95] we wrote

> *Slow [corrupted] actions can carry the effects of a transient error arbitrarily far into the future. Hence we can only say something meaningful about that part of a run after the time that all [corrupted] actions have finished, or the processors on which these [corrupted] actions run have crashed.*

Our definitions therefore included explicit reference to crash actions ψ_p, and we only required that actions not overlapping with corrupted actions would behave correctly. This is a rather weak requirement because it allows very slow processors to delay self-stabilizition indefinitely.

Luckily, our initial intuition turns out to be pessimistic. In fact, even though corrupted actions can behave arbitrary, they can be forced to do so consistently, in the sense that other actions have to agree on their (incorrect) behaviour. Thus even actions overlapping corrupted actions can be forced to behave according to the specification of the object (at the cost of using dual-reader registers, as we shall see later). This removes the need to explicitly mention the crash action in our model, keeping our definitions cleaner and closer to the original definition of linearizability [HW90, Her91]. It also makes our definitions stronger, resulting in stronger self-stabilizing objects.

Finally, in our 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.

3 Some impossibility results

The impossibility results to be presented in this section help to set the scene for the actual constructions of stabilizing shared registers in the sections to come. First we prove that stabilizing 1W1R safe bits[4] are not strong enough to implement k-stabilizing wait-free regular registers, if we restrict the writer to write a single bit. We conjecture that this impossibility remains even if the writer can write more than 1 bit. This immediately implies that stronger self-stabilizing objects, like atomic registers, cannot be implemented using such safe registers, either. Second, we show that 0-stabilizing 1WnR (multi-reader) atomic registers — and similar multi-user objects — cannot be constructed from 0-stabilizing 1W1R (single-reader) atomic registers.

We note that 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 processors' steps in a run.

Theorem 3.1 *There exists no deterministic implementation of a k-stabilizing wait-free 1W1R binary regular register using* 0*-stabilizing 1W1R binary safe sub-registers where the writer writes a single bit.*

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

[4] All objects considered in this paper are wait-free; for brevity we will not always explicitly mention this when referring to an object.

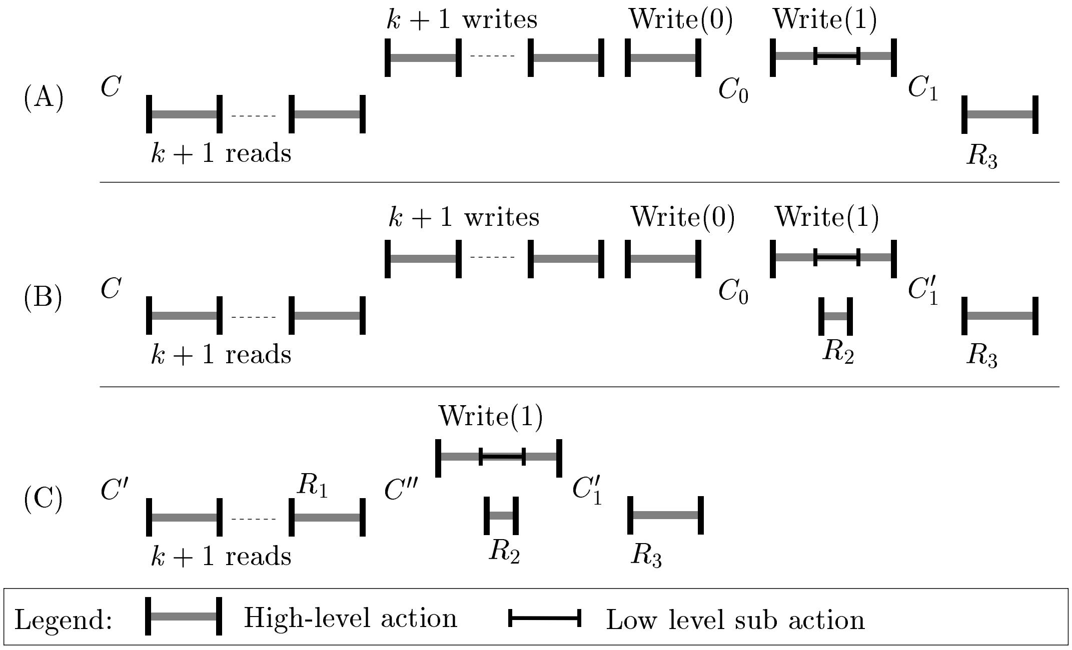

Figure 1: The runs constructed in the proof of Theorem 3.1.

Any implementation of a 1W1R binary regular register from stabilizing 1W1R safe binary sub-registers must use two sets of sub-registers (that can be considered as two "big" sub-registers): one (S_W) that is written by the writer and read by the reader and one (S_R) that is written by the reader and read by the writer. By assumption, S_W consists of a single bit. Then the whole state of the implementation is described by a configuration $C = (lr, lw, sr, sw)$, where lr, lw denote the reader's and writer's local states, and sr, sw denote the contents of S_R, S_W, respectively.

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. Then the value returned by each read is determined by a *reader function* $F(lr, sw)$. Furthermore, let $F^x(lr, sw)$ denote the value returned by the x-th read of a sequence of non-interfered reads that start from a configuration with local state lr while all reads find $\mathsf{S}_W = sw$. Let $C \Rightarrow v$ denote $F^{k+1}(C) = v$.

Consider an arbitrary initial configuration C such that $C \Rightarrow 1$. Schedule $k+1$ reads. The last read returns 1 by assumption. Next schedule $k+1$ writes. Then schedule a write of 0. Call the resulting state C_0. Because the run must be stabilised now, $C_0 \Rightarrow 0$. Then schedule a write of 1. Call the resulting state C_1. Now $C_1 \Rightarrow 1$, and in fact a read following the write of 1 must return 1. The complete execution appears in Fig. 1 (A)).

Clearly lr in C_0 equals lr in C_1. So to make sure that $C_1 \Rightarrow 1$ while $C_0 \Rightarrow 0$, the write of 1 must write and flip the bit in S_W. If this write writes the safe bit several times, consider the last time it does so. Call this action w.

Consider the same execution (A) as before, except that an extra read action is scheduled overlapping the write of 1, and in fact overlapping the last write to the safe bit w such that all reads of the safe bit are interfered by w. By the safeness properties of this bit and the fact that C_0 and C_1 differ only in the value of the single bit in S_W, this read can be made to observe exactly the same state as a read starting in C_0 in execution (A). This read must, by assumption, return 0. Call the state after both read and write have finished C'_1. Note that in C'_1, the state (both local and shared) of the reader may be different from C_1. Moreover, the write of 1 may, after its w action observe the interfering read and perform additional actions. Still $C'_1 \Rightarrow 1$, and in fact a read following the write of 1 must return 1. See Fig. 1 (B)).

Now let a transient error start the system in state C' where lw is taken from C_0, and sw, lr, sr are taken from C. Schedule again $k+1$ reads, leading to C''. These reads cannot distinguish C' from C, so the last read (called R_1) again returns 1 and lr and sr in C'' now have the same value as in C_0. Now schedule a write of 1. Call this write W. This write cannot distinguish C_0 from C'', so it will perform the same actions upto and including the action w that flips the bit in S_W. See Fig. 1 (C)).

Again schedule a read (called R_2) overlapping the write of 1, and in fact overlapping the sub action w such that all reads of the safe bit are interfered by w. By the safeness property, R_2 can be made to observe the same value for S_W as in C_0. Because in C'' the local state lr of the reader equals that of C_0, R_2 will return 0. Moreover, the resulting state after R_2 and W have finished is equal to C'_1 of execution (B). Now schedule another read R_3. By the last observation, R_3 will return 1.

We now have arrived at a contradiction. The only feasible write for R_3 is W. Because R_3 returns 1, W must have written 1. The only feasible write for R_1 equals the initial write, or one of the preceding reads. This feasible write also must have written 1. Because $\mathsf{count}(R_1)$ and $\mathsf{count}(R_2)$ are larger than k by construction, neither of them can behave as a write. Hence R_2 must return the value of a feasible write, which is either W or the write feasible for R_1. Both wrote 1, so R_2 should have returned 1. This is a contradiction. $\triangleleft$

We believe that such an implementation does not exist even if we allow the writer to write an arbitrary number of safe bits. The intuition behind this is as follows. Since the writer cannot read the sub-registers it writes, in order to know their contents and converge into correct stabilized behaviours, 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 same reasoning holds for the registers written by the reader. The adversary can set the system in a state in which this information is inconsistent. Subsequently, by scheduling the sub-actions of both the reader and the writer on the same sub-register to be concurrent, the adversary can destroy the information propagation because of the weak consistency that safeness guarantees.

Conjecture 3.2 *There exists no deterministic implementation of a k-stabilizing wait-free 1W1R binary regular register using 0-stabilizing 1W1R binary safe sub-registers.*

The next theorem shows that in order to implement a n-reader self-stabilizing register we must either

- settle for k-stabilization with $k > 0$, but using single reader registers in our construction, or
- achieve 0-stabilization, but using m-reader (e.g. dual reader), $m < n$, registers in our construction.

The construction of a 1-stabilizing wait-free nWnR atomic register from stabilizing 1W2R regular ones in Section 4.3 uses this approach.

Theorem 3.3 *For $n > 1$ and $z > 1$ there does not exist a deterministic implementation of a 0-stabilizing wait-free 1WnR z-ary atomic register from 0-stabilizing wait-free 1W1R atomic registers.*

Proof. Let all subregisters be 1W1R. Then reads of the atomic register executed by different processors must obtain a value by reading disjoint sets of subregisters. For every processor and each possible return value v of a read, the adversary can set the configuration of the processor (i.e., local state plus values read from the subregisters) independently such that if this processor executes its first read without any interference, this read returns v. Also note that each processor uses separate local variables and shared registers in the implementation of each atomic register, so that the adversary can set the configuration of each of these atomic registers independently.

Suppose, to the contrary, that such a 0-stabilizing wait-free implementation of a 1WnR z-ary atomic register A exists. W.l.o.g. ($z > 1$) assume that 0 and 1 are among the values stored in register A. Let p_0 and p_1 ($n > 1$) be two processors accessing A, and let the adversary set the configuration of p_0 such that its first read on its own will return 0, while the configuration of p_1 is such that its first read on its own will return 1. Consider all runs over A with only two actions: a read R_0 executed by p_0 and a read R_1 executed by p_1, both non-corrupted. Then according to Def. 2.5 in all such runs R_0 and R_1 must return the same value. Also, there is a run (the one where R_0 executes without interference) where 0 is the return value, and there is a run (the one where R_1 executes without interference) where 1 is the return value.

This construction now can be used to solve the two processor consensus problem as follows. Consider two copies A^0 and A^1 of the above 0-stabilizing wait-free implementation of a nWnR z-ary atomic register. Let the initial configuration of A^0 be such that read $R_0(\mathsf{A}^0)$ (of processor p_0) on its own returns 0, and the read $R_1(\mathsf{A}^0)$ (of processor p_1) on its own returns 1. Similarly, let the initial configuration of A^1 be such that read $R_0(\mathsf{A}^1)$ (of processor p_0) on its own returns 1, and the read $R_1(\mathsf{A}^1)$ (of processor p_1) on its own returns 0.

The protocol for p_i, where $i \in \{0, 1\}$, to propose a value $v \in \{0, 1\}$ simply is to read and return $R_i(\mathsf{A}^{\mathsf{i} \oplus \mathsf{v}})$, where $\oplus$ denotes the bitwise exclusive or. To see that this protocol solves 2-processor consensus consider the following two cases.

If p_0 and p_1 propose complementary values v and $1 \oplus v$ respectively, then both read the *same* register A^{v}, and by the observation of the second to last paragraph both must return the same value. This value is either v or $1 \oplus v$, both of which are proposed.

If p_0 and p_1 propose the same value v, they will *not* read the same register, and hence these reads will execute on their own. If p_i proposes v, it reads $R_i(\mathsf{A}^{\mathsf{i} \oplus \mathsf{v}})$ which, according to the initialisations described in the previous paragraph, will return v if executed on its own. Hence both processors decide v as required.

The above construction, although involving 0-stabilizing registers, does not involve any errors and is started in an initial state for each of the 1W1R 0-stabilizing atomic registers used to construct A^{i} with no corrupted actions pending on these registers. Any run over A^{i} induces a run over each of the 1W1R 0-stabilizing atomic registers, that, because of the above observation, corresponds to a run over an ordinary, non-selfstabilizing, 1W1R atomic

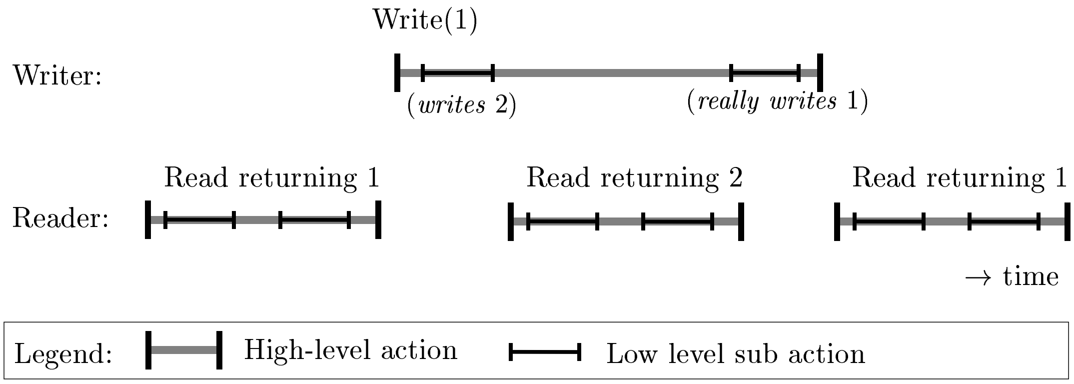

Figure 2: Repeating actions does not make an object 1-stabilizing.

register. Therefore, constructing A^{i} using such ordinary registers would result in the same behaviour in the described setting, and would, in particular, solve 2-processor consensus as well.

Loui and Abu-Amara [LAA87] showed that deterministically solving 1-resilient consensus (where only one processor may fail) using atomic read/write registers is impossible. We conclude that therefore a deterministic implementation from 0 stabilizing wait-free 1W1R atomic registers of a 0-stabilizing wait-free 1WnR z-ary atomic register does not exist. ◁

It is straightforward to generalise Theorem 3.3 and its proof to similar multi-user objects that are known not to be strong enough to solve consensus (e.g. regular registers and atomic snapshot memories [AAD$^+$90]). Also note that the proof of Theorem 3.3 would also hold for the construction of k-stabilizing registers, if the first k reads were allowed to behave arbitrary *but not as a write.* This shows that any such strengthening of Definition 2.5 (by imposing restrictions on the arbitrary behaviour of the first k actions of a processor) would be infeasible because implementations of such objects do not, by extensions to Theorem 3.3, exist.

We also would like to point out that a k-stabilizing shared object cannot simply be transformed into a 1-stabilizing one by executing each operation k times in a row. Consider the counterexample in Fig. 2 for $k = 2$. The first read (before the write W, and whose only feasible write is the initial write) returns 1. The third read (that starts after the write W, which constitutes its only feasible write) also returns 1. Hence the initial write and the write W both must have written 1. This contradicts the fact that the second read returns 2 (due to the fact that the first subwrite behaves arbitrary and writes 2 instead of 1), whose only feasible writes (write W and the initial write) both wrote 1.

4 Self-stabilizing constructions of shared registers

The results of the previous section raise the question whether any self-stabilizing shared objects can be built or do exist at all. We answer this question affirmatively, by giving constructions of regular and atomic self-stabilizing shared registers using *dual-reader* safe self-stabilizing registers, that allow the writer to read the values it writes to its own registers. We believe there is no fundamental difference between assuming that a 1W1R safe bit exists

S: stabilizing 1W2R `safe` bit

For $i \in \{0,1\}$:
operation $Read_i() : \{0,1\}$
return $Read_i(\mathsf{S})$;

operation $Write_0(v : \{0,1\})$
if $Read_0(\mathsf{S}) \neq v$
then $Write_0(\mathsf{S}, v)$;

Protocol 4.1: A 0-stabilizing 1W2R regular bit.

and assuming that a 1W2R safe bit exists. After all, the first models a flip-flop with a single output wire, whereas the latter models a flip-flop with its output wire split in two.

We therefore assume the existence of such 1W2R safe bits and use these as basic building blocks in the construction of a 1W2R 0-stabilizing regular bit (Section 4.1), a 1W2R z-ary 0-stabilizing regular register (Section 4.2), and a nWnR z-ary 1-stabilizing atomic register (Section 4.3). All these constructions are minor modifications of well-known constructions of the non-self-stabilizing, single-reader, equivalents. We do not present explicit constructions of single-reader self-stabilizing registers, because the dual-reader registers come more or less for free given a dual reader safe bit as basic building block. Moreover, we need dual-reader registers in the construction of the multi-writer atomic register.

A few words on the programming notation are in order. We use := to denote assignment, *name* to denote local variables, NAME to denote shared variables, and **name** for keywords.

4.1 A 0-stabilizing 1W2R regular bit

Protocol 4.1 presents the adaptation of Lamport's [Lam86] construction of a 1W1R regular bit from a 1W1R safe bit, into the construction of a 0-stabilizing wait-free 1W2R regular bit using a wait-free 0-stabilizing 1W2R safe bit. Instead of relying on a local copy of the value of the bit, the writer now actually reads the bit (over its second 'wire') to determine whether its value has to be changed. We proceed by proving correctness of the protocol.

Theorem 4.1 *Protocol 4.1 implements a wait-free 0-stabilizing 1W2R regular binary register using one wait-free 0-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. By Protocol 4.1 this induces an order $\rightarrow$ over the safe bit S. Let us use $R(\mathsf{S})$ ($W(\mathsf{S})$) to denote the read from (write to) the safe bit S performed by read R (write W) on the regular bit. Let $w_\perp$ be the initialising write of safe bit S, and set $\mathsf{val}(W_\perp) = \mathsf{val}(w_\perp)$ for the initialising write $W_\perp$ of the regular bit. If $\langle \mathcal{A}, \rightarrow \rangle$ has a corrupted write W, set $\mathsf{val}(W)$ to the value of S just after W; this is the value an interference free[5] read starting after W will read. In case of such a corrupted write, set $\mathsf{val}(w_\perp)$ (and $\mathsf{val}(W_\perp)$) to $\neg\mathsf{val}(W)$. For all other, non-corrupted, writes set $\mathsf{val}(W(v)) = v$. For all non-corrupted reads, set $\mathsf{val}(R) = \mathsf{val}(R(\mathsf{S}))$.

According to Def. 2.6 it remains to show that in $\langle \mathcal{A}, \rightarrow \rangle$ all non-corrupted reads R return the value written by a feasible write. Protocol 4.1 is trivially wait-free.

[5] A read is interference free if it does not overlap a write.

If a read overlaps a corrupted write W, then both W and $W_\perp$ are feasible writes for this read. By the definition of $\mathsf{val}(\cdot)$ above, $\mathsf{val}(W) \neq \mathsf{val}(W_\perp)$ and hence R always returns the value written by a feasible write.

To complete the proof we need the following claim.

Claim 4.2 *If $R(\mathsf{S})$ is interference free and not-corrupted and $W \rightleftharpoons R(\mathsf{S})$ then $\mathsf{val}(W) = \mathsf{val}(R(\mathsf{S}))$.*

Proof. Let $W \rightleftharpoons R(\mathsf{S})$ and let $\mathsf{val}(R(\mathsf{S})) = a$. If $W = W_\perp$ or W is a corrupted write, then $\mathsf{val}(W) = a$ by definition. Otherwise, note that a write $W(x)$ reads S by $R'(\mathsf{S})$ without interference. Either $\mathsf{val}(R'(\mathsf{S})) = x$ so W does not write S or $\mathsf{val}(R'(\mathsf{S})) = \neg x$ and W does write x to S by $W(\mathsf{S})$. Note that S is 0-stabilizing, and that none of these actions on S is corrupted.

In the first case, as there cannot be another write to S between $R'(\mathsf{S})$ and $R(\mathsf{S})$, $\mathsf{val}(R(\mathsf{S})) = \mathsf{val}(R'(\mathsf{S})) = x = \mathsf{val}(W)$. In the second case, as $W(\mathsf{S}) \rightleftharpoons R(\mathsf{S})$, $\mathsf{val}(R(\mathsf{S})) = \mathsf{val}(W(\mathsf{S})) = x = \mathsf{val}(W)$. ◁

Now consider a read R not overlapping a corrupted write. If $R(\mathsf{S})$ is interference-free, then by Claim 4.2, for a write W with $W \rightleftharpoons R(\mathsf{S})$ we have $\mathsf{val}(W) = \mathsf{val}(R(\mathsf{S})) = \mathsf{val}(R)$ and W is feasible for R.

If $R(\mathsf{S})$ is interfered, there is a write $W(x)$ with $W \| R$ writing S and W cannot be corrupted by assumption. W reads S by $R'(\mathsf{S})$ and $\mathsf{val}(R'(\mathsf{S})) = \neg x$. By Claim 4.2 and the fact that $R'(\mathsf{S})$ cannot overlap another write (because it is executed by one), for W' with $W' \rightleftharpoons W$ we must have $\mathsf{val}(W') = \neg x$. As $W \| R$, then $W' \rightleftharpoons R$ or $W' \| R$, so both W and W' are feasible for R. One of these writes writes 0 and the other writes 1, so $\mathsf{val}(R)$ equals the value written by a feasible write. ◁

4.2 A 0-stabilizing 1W2R z-ary regular register

Protocol 4.2 presents the adaptation of Lamport's [Lam86] construction of a 1W1R z-ary regular register from z regular 1W1R bits, into the construction of a wait-free 0-stabilizing 1W2R z-ary regular register using z wait-free 0-stabilizing 1W2R regular bits. It differs from the original construction on two points. First, a check is added to detect that the end of the bit array is reached without reading a 1 in any of the bits, in which case a default value $(z-1)$ is returned. Second, instead of using a while loop to zero some registers, a fixed programmed sequence of writes is used (see Section 2.1).

Suppose the writer would use the following code

while $v \neq 0$ **do** $v := v - 1$; $\mathit{Write}(\mathsf{S}^{\mathsf{v}}, 0)$;

to zero all registers $\mathsf{S}^{\mathsf{v}-1}, \ldots, \mathsf{S}^{0}$ that 'lie below' the written value v. Consider the following scenario (see also Figure 3), where a corrupted write writes 0 to S^{a} at the time of the error, but holds $v = c > a$ in its local state. After this write, it starts writing 0 to all $\mathsf{S}^{\mathsf{c}-1}, \ldots, \mathsf{S}^{0}$, writing 0 again to S^{a} some time later. Furthermore, assume that at the time of the error for some b, $a < b < c$ with $\mathsf{S}^{\mathsf{b}} = 1$, we have $\mathsf{S}^{\mathsf{j}} = 0$ for all $j \neq a$ with $0 \leq j < b$. Then a read R_3 started after the corrupted write (on the z-ary register) will read $\mathsf{S}^{\mathsf{i}} = 0$ for all $i < c$ and hence will return a value $v_3 \geq c$. A read R_1 started just after the error will read

$\mathsf{S}^0 \ldots \mathsf{S}^{z-1}$: stabilizing 1W2R `regular` bit

For $i \in \{0,1\}$:
operation $Read_i() : \{0,\ldots,z-1\}$
$w : \{0,\ldots,z\}$;
  $w := 0$;
  while $w < z \wedge Read_i(\mathsf{S}^{w}) = 0$
  do $w := w+1$;
  if $w = z$ **then return** $z-1$;
    else return w ;

operation $Write_0(v : \{0,\ldots,z-1\})$
$x : \{0,1\}$;
  $x := 1$;
  goto v
$z-1$: $Write_0(\mathsf{S}^{z-1}, x)$; $x := 0$;
$z-2$: $Write_0(\mathsf{S}^{z-2}, x)$; $x := 0$;
  $\vdots$
1: $Write_0(\mathsf{S}^{0}, x)$; $x := 0$;
0: **return**

Protocol 4.2: A 0-stabilizing 1W2R z-ary regular register.

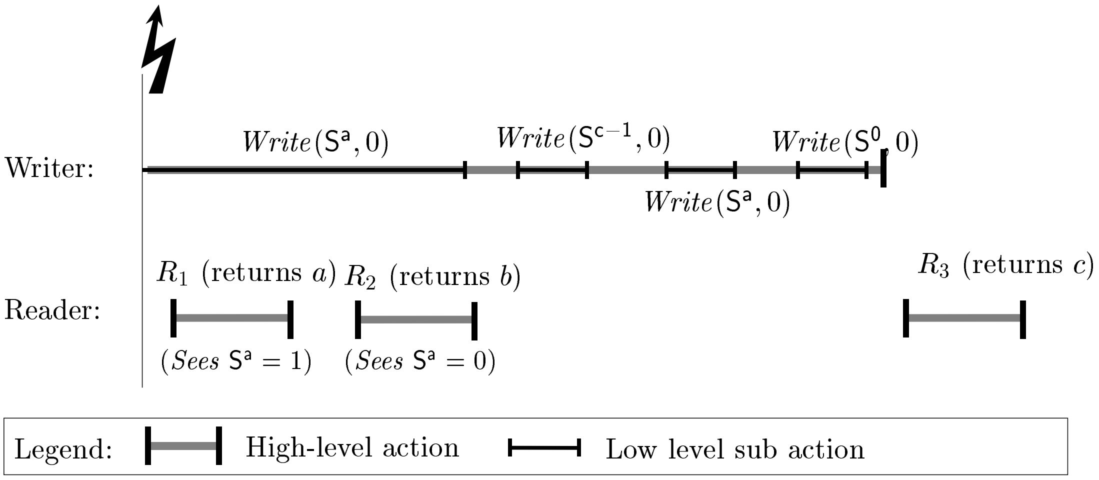

Figure 3: The danger of using a while loop.

$\mathsf{S}^i = 0$ for all $i < a$, may read $\mathsf{S}^a = 1$ (because the corrupted write is writing this register concurrently), and therefore may return $v_1 = a$. Another read R_2 started just after R_1 will also read $\mathsf{S}^i = 0$ for all $i < a$, may read $\mathsf{S}^a = 0$ (because the corrupted write is writing this register concurrently), and then will continue reading $\mathsf{S}^i = 0$ for all $a < i < b$, until it finds $\mathsf{S}^b = 1$ and thus returns $v_2 = b$. These three reads return three different values v_1, v_2 and v_3, while there are only two feasible writes $W_\perp$ (the initial write) and W^0 (the corrupted write) for these reads. This contradicts the 0-stabilizing regularity of the register.

4.2.1 Proof of correctness

Let $\langle \mathcal{A}, \rightarrow \rangle$ be an arbitrary run over the regular z-ary register. By Prot. 4.2 this induces an order $\rightarrow$ on the actions on the regular bits $\mathsf{S}^0, \ldots, \mathsf{S}^{z-1}$. Because there is only a single writer, we can number all the writes consecutively, writing W^i for the write with index $\mathsf{count}(W) = i$. Then W^0 is the corrupted write if it exists, and we set the index of $W_\perp$ (the initialising write of the z-ary register) to -1. Let us write $R(\mathsf{S}^v)$ for the read of S^v by read R, and let us write

$W(\mathsf{S}^\mathsf{v})$ for the write to S^v by a write W. We define the index of $W(\mathsf{S}^\mathsf{v})$ to equal the index of W. Note that this way, indices of consecutive writes to such a subregister may differ by more than 1. For non-corrupted reads R of a subregister S^v define $\pi(R)$ to be the largest index i such that W^i (of S^v) is feasible for R and we have $\mathsf{val}(W^i) = \mathsf{val}(R)$. This so-called *reading-function* is well defined because by 0-stabilizing regularity of S^v, such write always exists.

We now fix an assignment of $\mathsf{val}(W)$ for all write actions W and show that given this assignment, all non-corrupted reads return values written by feasible writes. For $W_\perp = W^{-1}$ we set $\mathsf{val}(W_\perp)$ to the minimal v such that $\mathsf{val}(W_\perp(\mathsf{S}^\mathsf{v})) = 1$, setting $\mathsf{val}(W_\perp) = z - 1$ if no such v exists. Suppose there is a corrupted write W^0. Let S^a be the first register written by this write. Then set $\mathsf{val}(W^0)$ to the minimal v such that $\mathsf{val}(W^0(\mathsf{S}^\mathsf{v})) = 1$ if $v \leq a$, and $\mathsf{val}(W_\perp(\mathsf{S}^\mathsf{v})) = 1$ if $v > a$, setting $\mathsf{val}(W^0) = z - 1$ if no such v exists. This way, $\mathsf{val}(W^0)$ equals the value read by an non-interfered read starting after W^0. For all other writes $W(v)$ set $\mathsf{val}(W(v)) = v$.

We first prove the following claim:

Claim 4.3 *Let R be a read reading $\mathsf{S}^{\mathsf{w}+1}$. If $\pi(R(\mathsf{S}^\mathsf{w})) > 0$ then $\pi(R(\mathsf{S}^\mathsf{w})) \leq \pi(R(\mathsf{S}^{\mathsf{w}+1}))$. If W^i with $i \geq 1$ is a write such that $W^i \to R$, then for all registers S^v read by R, we have $\pi(R(\mathsf{S}^\mathsf{v})) \geq i$*

Proof. If R reads $\mathsf{S}^{\mathsf{w}+1}$, then $\mathsf{val}(R(\mathsf{S}^\mathsf{w})) = 0$. If $\pi(R(\mathsf{S}^\mathsf{w})) = i > 0$ then $\mathsf{val}(W^i(\mathsf{S}^\mathsf{w})) = 0$ and hence (by Prot. 4.2 and the fact that W^i for $i > 0$ is not corrupted) W^i must write to $\mathsf{S}^{\mathsf{w}+1}$ as well. Now by $W^i(\mathsf{S}^\mathsf{w}) \not\leftarrow R(\mathsf{S}^\mathsf{w})$ and $W^i(\mathsf{S}^{\mathsf{w}+1}) \to W^i(\mathsf{S}^\mathsf{w})$ and $R(\mathsf{S}^\mathsf{w}) \to R(\mathsf{S}^{\mathsf{w}+1})$ we get $W^i(\mathsf{S}^{\mathsf{w}+1}) \to R(\mathsf{S}^{\mathsf{w}+1})$ and hence $i \leq \pi(R(\mathsf{S}^{\mathsf{w}+1}))$ which proves the result.

The second part easily follows using induction and the observation that $W^i \to R$ implies $\pi(R(\mathsf{S}^0)) \geq i$. ◁

Theorem 4.4 *Protocol 4.2 implements a 0-stabilizing 1W2R z-ary regular register using z 0-stabilizing 1W2R regular binary registers.*

Proof. According to Def. 2.6 we have to show that in every run $\langle \mathcal{A}, \to \rangle$ over the register all non-corrupted reads return the value written by a feasible write. Protocol 4.2 is trivially wait-free.

Consider a read R with $\mathsf{val}(R) = v$. Then R reads S^v and $\mathsf{val}(R(\mathsf{S}^\mathsf{v})) = 1$, unless $v = z-1$. Let $\pi(R(\mathsf{S}^\mathsf{v})) = i$. Then $W^i \not\leftarrow R$. If $\mathsf{val}(R(\mathsf{S}^\mathsf{v})) = 0$ (and hence $v = z - 1$), by the protocol, either $i = -1$ or $i = 0$ (because only W^0 or $W_\perp$ can possibly write 0 to $\mathsf{S}^{\mathsf{z}-1}$).

If $i > 0$ then $\mathsf{val}(W^i) = v$ — because W^i wrote 1 to S^v in this case. If $i = 0$, the construction of the writer is such that even the corrupted write W^0 writes a 1 to at most one register, and if it does, it will not later overwrite this 1. Therefore also $\mathsf{val}(W^0) = v$. And if W^0 writes $\mathsf{S}^{\mathsf{z}-1}$ with $\mathsf{val}(W^0(\mathsf{S}^{\mathsf{z}-1})) = 0$ then it also writes 0 to all other registers and hence $\mathsf{val}(W^0) = z - 1$ by definition.

In both cases, W^i is not a feasible write for R only if there exists a W^j such that $W^i \to W^j \to R$ (and so $i < j$ and in particular $j > 0$). But then by Claim 4.3 $\pi(R(\mathsf{S}^\mathsf{v})) \geq j > i$, a contradiction.

The only case that remains is $i = -1$. By Claim 4.3, then $\pi(R(\mathsf{S}^\mathsf{w}))$ equals 0 or -1 for all $w < v$ (or else $\pi(R(\mathsf{S}^\mathsf{v})) > -1$). Now there are two cases.

If $\pi(R(\mathsf{S}^{\mathsf{w}})) = -1$ for all $w < v$, then $\mathsf{val}(W^{-1}(\mathsf{S}^{\mathsf{w}})) = 0$ for $w < v$. Hence $v = \mathsf{val}(W^{-1})$ and so $\mathsf{val}(R) = \mathsf{val}(W^{-1})$. Moreover, $R \not\leftarrow W^i$ for $i \geq 0$ (or else $\pi(R(\mathsf{S}^{\mathsf{w}})) \geq 0$ for some w) so W^{-1} is feasible for R.

If $\pi(R(\mathsf{S}^{\mathsf{w}})) = 0$ for some $w < v$, then we have $W^0(\mathsf{S}^{\mathsf{w}}) \not\leftarrow R(\mathsf{S}^{\mathsf{w}})$ Consider the first register S^{a} written by W^0. Clearly $w \leq a$. Moreover, if $w < a$, then using $R(\mathsf{S}^{\mathsf{w}}) \rightarrow R(\mathsf{S}^{\mathsf{a}})$ and $W^0(\mathsf{S}^{\mathsf{a}}) \rightarrow W^0(\mathsf{S}^{\mathsf{w}})$ we have $W^0(\mathsf{S}^{\mathsf{a}}) \rightarrow R(\mathsf{S}^{\mathsf{a}})$ and hence $\pi(R(\mathsf{S}^{\mathsf{a}})) = 0$. We have by assumption that $\pi(R(\mathsf{S}^{\mathsf{v}})) = i = -1$, so R continues to read past register S^{a}. Then it reads 0 from S^{a}, and hence $\mathsf{val}(W^0(\mathsf{S}^{\mathsf{a}})) = 0$. Because W^0 always writes 0 to all registers except perhaps the first it writes, $\mathsf{val}(W^0(\mathsf{S}^{\mathsf{i}})) = 0$ for all $i < a$ as well. For all w with $a < w \leq v$ we have $\pi(R(\mathsf{S}^{\mathsf{w}})) = -1$. Hence $\mathsf{val}(R) = v$ corresponds to the minimal v such that $\mathsf{val}(W^0(\mathsf{S}^{\mathsf{v}})) = 1$ if $v \leq a$, and $\mathsf{val}(W_\perp(\mathsf{S}^{\mathsf{v}})) = 1$ if $v > a$, setting $\mathsf{val}(W^0) = z - 1$ if no such v exists. In other words, $\mathsf{val}(R) = \mathsf{val}(W^0)$. Finally note that $R \not\leftarrow W^i$ for $i \geq 1$ (or else $\pi(R(\mathsf{S}^{\mathsf{w}})) \geq 1$ for some w), so W^0 is feasible for R. $\triangleleft$

This completes our proof of correctness for Protocol 4.2

4.3 A 1-stabilizing nWnR z-ary atomic register

Protocol 4.3 presents the adaptation of the Vitányi-Awerbuch [VA86, AKKV88] multi-writer atomic register construction from 1W1R multi-valued regular registers, into a wait-free 1-stabilizing nWnR z-ary atomic register using n^2 wait-free 0-stabilizing 1W2R ∞-ary regular registers.

To make this construction self-stabilizing the values have become part of the labels used to determine the most recently written value. In the original construction this is not necessary because any value stored with a tag, processor id pair always corresponds to the value written by that processor during that invocation. Moreover, every label written in the propagation phase is read back and used in the next write. This prevents a corrupted write from injecting two different bad values in the register (one by the corrupted subwrite and the other by the erroneous value of the current label to be written). Finally, the propagation phase only terminates if the validation phase has made certain that all registers that have to be written indeed contain the same value. Clearly this check is superfluous for almost all cases, except when we consider corrupted actions. In that case, the check ensures that if the corrupted action writes any register at all, then it will continue writing registers until all registers contain the same value (see 4.5).

The main difficulty — and difference — in the proof of correctness exists in showing that the first few actions of a processor behave in line with Def. 2.4. We prove correctness of the protocol below.

In the protocol, $\mathcal{V}$ is the domain of values read and written by the multi-writer register. The construction uses n^2 regular 0-stabilizing registers S_{ij} written by processor i and read by processor i and 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 $\leq$, and $\perp$ is the minimal label.

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 complete run of the above protocol.

$\mathsf{S}_{11} \ldots \mathsf{S}_{nn}$: 0-stabilizing 1W2R
`regular`: $I\!N \times \{1,\ldots,n\} \times \mathcal{V}$
(with fields *tag*, *id*, and *val*)

For $i \in \{1,\ldots,n\}$:
function $ReadCur_i() : \mathcal{V}$
$cur : I\!N \times \{1,\ldots,n\} \times \mathcal{V}$;
  $cur := \perp$;
  $cur := \max(cur, Read_i(\mathsf{S}_{1i}))$;
  $cur := \max(cur, Read_i(\mathsf{S}_{2i}))$;
  $\vdots$
  $cur := \max(cur, Read_i(\mathsf{S}_{ni}))$;
  return cur ;

operation $Read_i() : \mathcal{V}$
$cur : I\!N \times \{1,\ldots,n\} \times \mathcal{V}$
  $cur := ReadCur_i()$;
  $Propagate_i(cur)$;
  return $cur.val$;

function $Validate_i(v : \mathcal{V}) : \{true, false\}$
  return ($Read_i(\mathsf{S}_{i1}) = v$
    $\wedge\ Read_i(\mathsf{S}_{i2}) = v$
    $\vdots$
    $\wedge\ Read_i(\mathsf{S}_{in}) = v)$;

function $Propagate_i(v : \mathcal{V})$
  repeat $Write_i(\mathsf{S}_{i1}, v)$; $v := Read_i(\mathsf{S}_{i1})$;
    $Write_i(\mathsf{S}_{i2}, v)$; $v := Read_i(\mathsf{S}_{i2})$;
    $\vdots$
    $Write_i(\mathsf{S}_{in}, v)$; $v := Read_i(\mathsf{S}_{in})$;
  until $Validate_i(v)$

operation $Write_i(v : \mathcal{V})$
$cur, new : I\!N \times \{1,\ldots,n\} \times \mathcal{V}$
  $cur := ReadCur_i()$;
  $new := \langle cur.tag + 1, i, v\rangle$;
  $Propagate_i(new)$;

Protocol 4.3: A 1-stabilizing nWnR z-ary atomic register.

Lemma 4.5 *Let $A \in \mathcal{A}$ be an action executed by processor i. If A writes at least one register S_{ij} for some j, then at the end of A we have $(\forall j,k :: \mathsf{S}_{ij} = \mathsf{S}_{ik})$*

Proof. Looking at Protocol 4.3, we see that every write to a register is (eventually) followed by a call to $Validate_i(v)$ which only returns *true* (allowing the action to terminate) if $(\forall j :: \mathsf{S}_{ij} = v)$. $\lhd$

Now remove from $\mathcal{A}$ all actions that do not write a single register as a subaction. This can only affect corrupted actions — all other actions write at least n registers. We will show that the resulting run $\langle \mathcal{A}, \rightarrow\rangle$ is linearisable to $\langle \mathcal{A}, \Rightarrow\rangle$. The corrupted actions we remove can all be considered reads returning arbitrary values, and can be ordered before all other actions in $\langle \mathcal{A}, \Rightarrow\rangle$.

Now define for $A \in \mathcal{A}$, $\mathsf{label}(A)$ as the label written by the last subwrite performed by A. For all correct reads R, $\mathsf{label}(R)$ corresponds to the value returned by the read, and for all correct writes W, $\mathsf{label}(W)$ corresponds to the value written. Then the following is an immediate corollary of Lemma 4.5 (recall that we have removed all corrupted actions from the run that do not write any register)

Corollary 4.6 *For all $A \in \mathcal{A}$ performed by i, we have $(\forall j :: \mathsf{S}_{ij} = \mathsf{label}(A))$ immediately after termination of A.*

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

$$\mathcal{F} \;=\; \{A \in \mathcal{A} \mid \mathsf{count}(A) \leq 1\} \;,$$

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

Then $\mathcal{F}$ corresponds to the set of actions that, according to Def. 2.4, 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 set of actions $\mathcal{F}$, $\mathsf{label}(\mathcal{F}) = \{\mathsf{label}(F) \mid F \in \mathcal{F}\}$. Set $\Lambda = \mathsf{label}(\mathcal{F}) \backslash \mathsf{label}(\mathcal{W}^-)$, the set of labels not written by a real write in $\mathcal{W}^-$, and let $\mathcal{F}_W$ be an arbitrary subset of $\mathcal{F}$ such that

($F1$) $\mathsf{label}(\mathcal{F}_W) = \Lambda$,

($F2$) 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 $\mathcal{R} = \mathcal{R}^- \cup \mathcal{F}_R$.

Note that we can construct a set $\mathcal{F}_W$ satisfying these conditions by considering in turn all actions in $\mathcal{F}$ ordered by their unique invocation times, and adding to $\mathcal{F}_W$ every action whose label is not yet in $\mathcal{F}_W$.

Lemma 4.7 *If $A \to B$ 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. By Corollary 4.6, after A we have $\mathsf{S}_{ij} = \mathsf{label}(A)$. If $A \to B$, either B reads $\mathsf{label}(A)$ from S_{ij} (because it is a 0-stabilizing register) or a later subwrite to S_{ij} by some action C of i is a feasible write to the subread of S_{ij} of B. In the first case, because B picks the maximum of the values read, $\mathsf{label}(A) \leq \mathsf{label}(B)$

In the second case $\mathsf{label}(C) \leq \mathsf{label}(B)$ and there is a sequence of actions C_1, C_2 etc. executed by i between A and C, i.e. $A = C_0 \to C_1 \to C_2 \ldots \to C$. By Corollary 4.6, after C_i we have $\mathsf{S}_{ii} = \mathsf{label}(C_i)$ as well. Because C_{i+1} reads S_{ii}, we have $\mathsf{label}(C_i) \leq \mathsf{label}(C_{i+1})$. We conclude $\mathsf{label}(A) \leq \mathsf{label}(C)$ and therefore $\mathsf{label}(A) \leq \mathsf{label}(B)$.

In either case, if B is a write, then it increases the tag by one and hence $\mathsf{label}(A) < \mathsf{label}(B)$. ◁

The next lemma basically shows that every read returns a value written by some write. Uniqueness of this write is established in Lemma 4.9.

Lemma 4.8 *For all $R \in \mathcal{R}$ there exists a $W \in \mathcal{W}$ such that $\mathsf{label}(W) = \mathsf{label}(R)$ and $W \not\leftarrow R$.*

Proof. Consider actions A executed by processor i and actions B executed by processor j. Let $W(A, B)$ be the last subwrite to S_{ij} by action A, and let $R(B, A)$ be the first subread from S_{ij} by action B. Define $A \hookrightarrow B$ iff $\mathsf{label}(A) = \mathsf{label}(B)$ and $W(A, B) \not\leftarrow R(B, A)$. Note that for all A, B with $\mathsf{count}(\cdot) > 1$ we cannot have both $A \hookrightarrow B$ and $B \hookrightarrow A$ because all reads by A and B precede their writes (See Protocol 4.3). In other words, $A \hookrightarrow B$ if B

could have copied its label from A. Clearly $A \hookrightarrow B$ implies $A \not\leftarrow B$. Similarly $A \to B$ and $\mathsf{label}(A) = \mathsf{label}(B)$ implies $A \hookrightarrow B$, because then $W(A, B) \not\leftarrow R(B, A)$.

Now let $R \in \mathcal{R}$ be arbitrary, and pick a $B \in \mathcal{A}$ such that $B \hookrightarrow R$ and for no $A \in \mathcal{A}$, $A \hookrightarrow B$. If no such B exists, set $B = R$. 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 $C \to B$. If $\mathsf{label}(C) = \mathsf{label}(B)$ then $C \hookrightarrow B$, while if $\mathsf{label}(C) < \mathsf{label}(B)$ (the only other possible case according to Lemma 4.7) then the contents of the register from which B obtains $\mathsf{label}(B)$ — note that since B is a read it writes the maximal label it reads — 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 $D \hookrightarrow B$. This contradicts the assumption that there is no A such that $A \hookrightarrow B$.

We conclude that $\mathsf{count}(B) \leq 1$ and hence $B \in \mathcal{F}$. This implies $\mathsf{label}(B) \in \mathsf{label}(\mathcal{F})$. So either there exists a $W \in \mathcal{W}^-$ such that $\mathsf{label}(W) = \mathsf{label}(B) = \mathsf{label}(R)$, or $\mathsf{label}(B) \in \Lambda$ 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 4.7, $W \not\leftarrow R$ as required. In the second case, since $B \in \mathcal{F}$ we must have by $(F3)$, $t_I(W') < t_I(B)$. Then as $B \hookrightarrow R$ implies $B \not\leftarrow R$, this in turn implies $W' \not\leftarrow R$. ◁

The next lemma shows that different write actions write different labels. Together with the previous lemma this establishes that the maximal label read by a read action uniquely determines the write action that wrote the value this read returns.

Lemma 4.9 *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 Prot 4.3 W and W' must be executed by the same processor (or else their *id*-fields differ). If $\mathsf{label}(W) = \mathsf{label}(W')$, and using Lemma 4.7, neither $W \to W'$ nor $W' \to W$. Hence $W = W'$.

$W \in W^-, W' \in \mathcal{F}_W$**:** Then $\mathsf{label}(W) \notin \Lambda$ by definition of Λ, and $\mathsf{label}(W') \in \Lambda$ by $(F1)$. This is a contradiction.

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

Define for a particular run $\langle \mathcal{A}, \to \rangle$ a *reading function* $\pi : \mathcal{R} \mapsto \mathcal{W}$ by $\pi(R) = W$ if $\mathsf{label}(R) = \mathsf{label}(W)$ and $W \in \mathcal{W}$. This is a proper definition by the next lemma.

Lemma 4.10 *For all $R \in \mathcal{R}$, $\pi(R)$ is defined and unique, $R \not\twoheadrightarrow \pi(R)$, and R returns the value written by $\pi(R)$.*

Proof. That $\pi(R)$ is defined and $R \not\twoheadrightarrow \pi(R)$ follows from Lemma 4.8. That it is unique follows from Lemma 4.9. If $\pi(R) \in \mathcal{W}^-$, then (stretching notation somewhat) $\mathsf{label}(\pi(R)).val$ 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 $\langle \mathcal{A}, \to \rangle$ 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 $\to'$ over Γ by

$$[W] \to' [W'] \iff (\exists A \in [W], B \in [W'] :: A \to B) \quad .$$

Lemma 4.11 *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 4.9. $\triangleleft$

Lemma 4.12 *$\to'$ is an acyclic partial order over Γ.*

Proof. Suppose not. Then there exists a chain

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

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 $A_i \to B_{i+1}$ (where $m + 1 = 1$). By Lemma 4.7 and 4.11 $\mathsf{label}(A_i) \leq \mathsf{label}(B_{i+1}) = \mathsf{label}(A_{i+1})$. We conclude that $\mathsf{label}(A_1) = \mathsf{label}(A_2)$, contrary to Lemma 4.11. $\triangleleft$

Applying these lemmas and the results of [AKKV88] we arrive at the following theorem.

Theorem 4.13 *Protocol 4.3 implements a 1-stabilizing nWnR z-ary atomic register using n^2 stabilizing 1W1R ∞-ary regular registers.*

Proof. Define a total order $\Rightarrow$ over $\mathcal{A}$ extending $\to$ as follows. First extend $\to'$ over Γ to a total order $\Rightarrow'$ (according to Lemma 4.12, this is possible). Now for $A \in [W]$ and $B \in [W']$ let $A \Rightarrow B$ if $[W] \Rightarrow' [W']$ (a). This extends $\to$ because if $A \to B$, then by the definition of $\to'$, $[W] \to' [W']$ and thus $[W] \Rightarrow' [W']$. For $A, B \in [W]$ fix an arbitrary extension $\Rightarrow$ of $\to$ 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 $\to$ because by Lemma 4.10, $C \not\to W$.

Now $\Rightarrow$ is a total order over all actions $\mathcal{A}$ such that for all $R \in \mathcal{R}$ we have $\pi(R) \Rightarrow R$ by Lemma 4.10 and (b). Moreover, 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 4.11, either $W \Rightarrow [\pi(R)]$ or $R \Rightarrow [W]$. Hence $W \Rightarrow \pi(R)$ or $R \Rightarrow W$.

We conclude that all reads R return the value written by the most recently preceding write in the sequential execution $\langle \mathcal{A}, \Rightarrow \rangle$. $\triangleleft$

5 Conclusions and Further research

Our results are a first, but important, step towards exploring the relation between self-stabilization and wait-freedom in the construction of shared objects. This is a new area of research and there are still a lot of interesting questions in this new area that remain unanswered. First of all, this paper is a proposal for a reasonable and general definition of self-stabilizing wait-free shared objects. The impossibility proofs and the constructions presented here are evidence for the viability of our approach. They show that our definitions are neither trivial nor impractical, but further research is necessary to assess their true value. In particular, the question is whether shared memory objects exist that only have k-stabilizing implementations for $k > 1$ (or even $k > 0$).

Second, we would like to prove or conjecture that single reader safe bits are in general not strong enough to implement a self-stabilizing regular register.

Third, the construction of the 1-stabilizing nWnR atomic register uses unbounded timestamps to invalidate old values. We would like to know whether this is necessarily so, or if the space requirements of a k-stabilizing atomic register can be bounded. Moreover, we would like to know whether we can close the gap between Protocol 4.3 and Theorem 3.3. I.e., whether it is possible either to construct a 1-stabilizing nWnR atomic register using *single* reader 0-stabilizing regular registers, or to construct a 0-stabilizing nWnR atomic register using 0-stabilizing 1W1R regular registers.

Fourth, the ramifications of – so far – having only 1-stabilizing atomic registers available for communication in self-stabilizing protocols should be investigated further.

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. In particular, we would be interested to know whether there are general methods to decrease the stabilization delay of a shared memory object from k to $k' < k$.

Acknowledgements

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

Due to unforeseen circumstances we were not able to publish our initial results [HPT95] in a thorough and more polished form any earlier. We apologise for this delay and the inconveniences it may have caused.

References

[ALR88] A. Lynch nancy, and R.Tuttle, M. An introduction to input/output automata. *CWI Quarterly* **2**, 3 (1988), 219–246.

[AAD+90] Afek, Y., Attiya, H., Dolev, D., Gafni, E., Merritt, M., and Shavit, N. Atomic snapshots of shared memory. In *9th Ann. Symp. on Principles of Distributed Computing* (Quebec City, Qu., Canada, 1990), ACM Press, pp. 1–13.

[AGMT92] Afek, Y., Greenberg, D. S., Merritt, M., and Taubenfeld, G. Computing with faulty shared memory. In *11th Ann. Symp. on Principles of Distributed Computing* (Vancouver, B.C., Canada, 1992), ACM Press, pp. 47–58.

[AMT93] Afek, Y., Merritt, M., and Taubenfeld, G. Benign failure models for shared memory. In *7th Int. Workshop on Distributed Algorithms* (Lausanne, Switzerland, 1993), A. Schiper (Ed.), Lect. Not. Comp. Sci. 725, Springer-Verlag, pp. 69–83.

[AH93] Anagnostou, E., and Hadzilacos, V. Tolerating transient and permanent failures. In *7th Int. Workshop on Distributed Algorithms* (Lausanne, Switzerland, 1993), A. Schiper (Ed.), Lect. Not. Comp. Sci. 725, Springer-Verlag, pp. 174–188.

[AH90] Aspnes, J., and Herlihy, M. P. Wait-free data structures in the asynchronous PRAM model. In *2nd Ann. Symp. on Parallel Algorithms and Architectures* (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 Conf. on Foundations of Software Technology and Theoretical Computer Science* (Pune, India, 1988), K. V. Nori and S. Kumar (Eds.), Lect. Not. Comp. Sci. 338, Springer Verlag, pp. 286–303.

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

[DW93] Dolev, S., and Welch, J. L. Wait-free clock synchronization. In *12th Ann. Symp. on Principles of Distributed Computing* (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 Ann. Symp. on Principles of Distributed Computing* (Ithaca, NY, USA, 1993), ACM Press, pp. 195–206.

[Her91] Herlihy, M. P. Wait-free synchronization. *ACM Transactions on Programming Languages and Systems* **13**, 1 (1991), 124–149.

[HW90] Herlihy, M. P., and Wing, J. M. Linearizability: A correctness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems* **12**, 3 (1990), 463–492.

[HPT95] Hoepman, J.-H., Papatriantafilou, M., and Tsigas, P. Self-stabilization of wait-free shared memory objects. In *9th Int. Workshop on Distributed Algorithms* (Le Mont-Saint-Michel, France, 1995), J.-M. Hélary and M. Raynal (Eds.), Lect. Not. Comp. Sci. 972, Springer, pp. 273–287.

[IS92] Israeli, A., and Shaham, A. Optimal multi-writer multi-reader atomic register. In *11th Ann. Symp. on Principles of Distributed Computing* (Vancouver, B.C., Canada, 1992), ACM Press, pp. 71–82.

[JCT92] Jayanti, P., Chandra, T. D., and Toueg, S. Fault-tolerant wait-free shared objects. In *33rd Symp. on Foundations of Computer Science* (Pittsburgh, PA, USA, 1992), IEEE Comp. Soc. Press, pp. 157–166.

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

[LV91] Li, M., and Vitányi, P. M. B. Optimality of wait-free atomic multiwriter variables. Tech. Rep. CS-R9128, Stichting Mathematisch Centrum (CWI), Amsterdam, 1991.

[LAA87] Loui, M. C., and Abu-Amara, H. H. Memory requirements for agreement among unreliable asynchronous processes. In *Advances in Computing Research*, F. P. Preparata (Ed.), vol. 4. JAI Press, 1987, pp. 163–183.

[PT94] Papatriantafilou, M., and Tsigas, P. On self-stabilizing wait-free clock synchronization. In *4th Scandinavian Workshop on Algorithm Theory* (Århus, Denmark, 1994), Lect. Not. Comp. Sci. 824, Springer Verlag, pp. 267–277.

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

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