Chapter 3

Safe, regular and atomic registers

October 1, 2007

1 Introduction

This part of the book is devoted to the construction of the simplest linearizable objects that are usually considered, namely shared *storage* objects that provide their users with two basic operations: *read* and *write*. These objects are usually called *registers*, and linerarizable registers are called *atomic registers*. In particular, we focus on how to wait-free implement such atomic registers using "weaker" registers. Again, the picture to have in mind is one where the weak registers are provided in hardware and the strongest registers are emulated in software to facilitate the job of the application's developer.

This chapter defines different sorts of registers and these differences depend on three dimensions: (a) the capacity of a register, (b) the access pattern to a register and (c) the behavior of a register in face of concurrency. The capacity of a register conveys the range of values it can store and we will in particular distinguish registers that can store a binary value from those than can store any number (possibly an infinite number) of values. The access pattern or a register conveys the number of processes that can read (resp. write) in a register. Finally, we will distinguish registers that do not provide any guarantee if accessed concurrently at one extreme, from those that ensure atomicity at the other extreme. The weakest kind of a shared register is one that can store a binary value, can be read by a single process p, can be written by a single process q, and does not ensure guarantee on the value read by p when p and q access it concurrently. We will show how, using several of such registers, we can build an atomic register that can store an arbitrary number of values and be read and written by any number of processes. This construction will be presented incrementally, going through intermediate kinds of registers, themselves interesting in their own rights.

An algorithm used to implement a register of a given kind from one with another kind is sometimes called *transformation* or *reduction*, the first high-level register being "reduced" to the second register used as a base object in the implementation. We also say that the high-level register is emulated by the second one.

2 The many faces of registers

The capacity of a register At any given time, a register contains a single value. According to the write operations issued by the processes, the register can contain distinct values at different times. So, the first dimension that characterizes a register is related to its size, i.e., its capacity to contain more or less information.

The simplest kind of register is the *binary* register: it can only store a single bit 0 or 1. We talk about a *shared bit*, or simply a *bit*.

Otherwise, a register is *multi-valued*. A multi-valued register can be bounded or unbounded. A *bounded* register is one whose value domain contains exactly b distinct values (e.g., the values from 0 until $b-1$) where b is typically a constant known by the processes. Otherwise the register is said to be unbounded.

A register that can contain b distinct values is said to be *b-valued*. Its binary representation requires $B = \lceil \log_2 b \rceil$ bits. Its unary representation is more expensive as it requires b bits (the value v being then represented with a bit equal to 1 followed by $v-1$ bits equal to 0).

The access pattern to a register This dimension concerns the number of processes that can read from or write into the register. A register is called *single-writer*, denoted SW, (resp., *single-reader*, noted SR) if only one specific process, known in advance, and called the *writer* (resp., the *reader*) can invoke the write (resp., read) operation on the register. A register that can be written (resp., read) by any process is called a *multi-writer* (resp., *multi-reader*) register. Such a register is denoted MW (resp., MR).

For instance, a binary SWMR register is a register that (a) can contain only 0 or 1, (b) can be read by all the processes but (c) written by a single process.

The concurrent behavior of a register When accessed sequentially, the behavior of a register is simple to define: a read invocation returns the last value written. When accessed concurrently, the semantics is more involved and several variants have been considered. We overview these variants in the next section.

3 Safe, regular and atomic registers

We overview three kinds of registers that vary according to their behavior in presence of concurrent accesses. They differ in the value returned by a read operation that access the register concurrently with a write operation. So, we always have the following:

- A read that is not concurrent with a write (i.e., their executions do not overlap) returns the current value of the register.

3.1 Safe registers

A *safe* register is the weakest traditionally considered in distributed computing. It always has a single writer. In this sense, the problem of concurrent write operations is a priori eliminated. Given that a read does not impact any other concurrent read, the only concurrency issue that needs to be addressed concerns the value returned by a read operation that is concurrent with a write.

- A read that is concurrent with one or several writes, returns any element of the value domain of the register.

It is important to see that, in presence of concurrency, the value returned by a read operation can possibly be a value that has never been written. It is constrained only by being within the value domain of the safe register. To illustrate this, consider a safe register that can contain only $b = 3$ values, e.g., $1, 2$ and 3. The register is bounded. Assuming that the current value is 1, consider a write of value 2 that is concurrent with a read operation. That read operation can returns 1, 2 or even 3. It cannot return 5 as that value is not in the domain of the safe register.

An interesting particular case is the binary SWMR safe register. This can contain only 0 and 1 and can be seen as modeling a flickering bit. Whatever its previous value, the value of the register can flicker during a write operation and only when the write finishes the register stabilizes to its final value (the value just written) and keep that value until the next write.

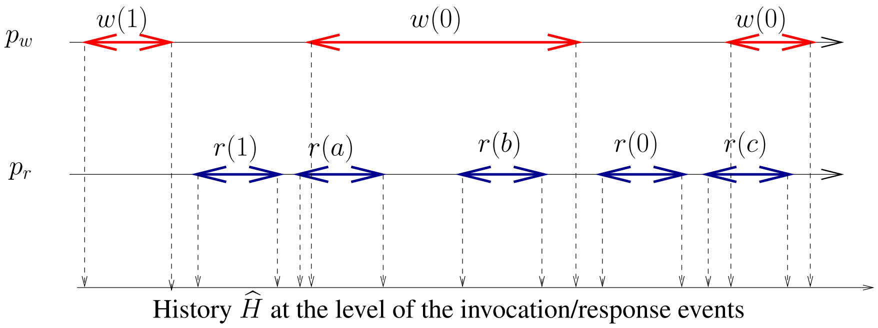

Figure 1: History of a register

Value returned	a	b	c
Safe	$1/0$	$1/0$	$1/0$
Regular	$1/0$	$1/0$	0
Atomic	1	$1/0$	0
	0	0	0

Table 1: Safe, regular and atomic registers

An example of the behavior of a binary safe register (i.e., a safe bit) is depicted in Figure 1 and Table 1. We consider there a SRSW safe register: only one reader is involved. The writer process is denoted p_w whereas the reader process is denoted p_r ($w(v)$ stands for a write operation that writes the value v; similarly, $r(v)$ stands for a read operation that obtains the value v). As the first and the fourth read operations do not overlap a write, they return the current value of the register, namely, 1 for the first read and 0 for the fourth one. The values returned by the other read operations are denoted a, b and c. All these read operations overlap a write and can consequently return any of the values that the register can contain (this is denoted $1/0$ as the register is binary in Table 1). So, the last read can return 1 even if the previous value was 0 and the concurrent operation writes the very same value 0. This gives 8 possible correct executions, assuming indeed a binary safe register.

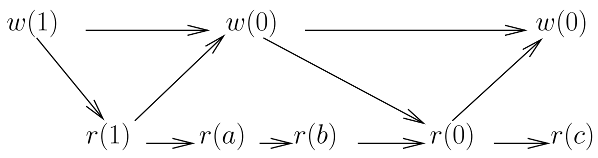

Figure 2: History of a safe register

Figure 2 depicts the corresponding history at the operation level (i.e., the partial order on the operations denoted $\rightarrow_H$). The transitive dependencies are not indicated. The unordered operations (e.g., the second $w(0)$ operation issued by p_w and $r(c)$ issued by p_r) are concurrent.

3.2 Regular registers

A *regular* register is also defined for the case of a single writer. It is a safe register that satisfies the following additional property:

- A read that is concurrent with one or several write operations returns the value of the register before these write operations, or the value written by any of these write operations.

As we can see, a value returned by a read is always a value that has been written either by the immediately preceding write, or by a concurrent write.

To illustrate the regular register notion, let us again consider Figure 1. The possible values returned are described in Table 1. The second read operation can return either the previous value or the value of the concurrent write, namely, 0 or 1. It is the same for the third read operation. In contrast, as the last write does not change the value of the register, the last read can return only the value 0. This means that 4 possible correct executions can be determined for Figure 1.

It is important to see that a read that overlaps several write operations can return any value among the values written by these writes and the value of the register before these writes. This is depicted in Figure 3 where value a returned by the second read can be any of 1, 2, 3, 4 or 5.

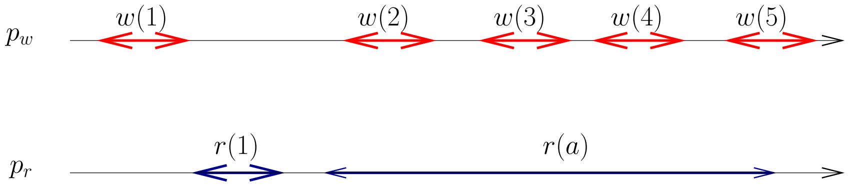

Figure 3: History of a regular register

3.3 Atomic registers

An *atomic* register is a MWMR register whose execution histories are linearizable. This means that it is possible to totally order all its read and write operations in such a way that this total order $\widehat{S}$ respects their real-time occurrence order and each read returns the value written by the last write operation that precedes it in $\widehat{S}$ (legality property).

Let us again consider Figure 1 to illustrate the atomicity notion for the specific case of a register. The second read is concurrent with the $w(0)$ operation. Given that the previous value of the register is 1, the second read can return either 1 or 0. If it returns 1 (the previous value of the register), the third read can return either 1 or 0. In contrast, if the second read returns 0 (the value written by the concurrent write), only value 0 can be returned by the third read as the second read indicates that the value 1 is now overwritten by the new value 0. Finally, the last read can only return the value 0. It is easy to see that there are 3 possible executions when the registers are binary and atomic. As previously, the possible values returned by the three read operations concurrent with a write operation are summarized in Table 1.

3.4 From regularity to atomicity: a theorem

It is interesting to notice that, when we look at the execution depicted in Figure 1 and Table 1, the only difference between a regular execution and an atomic execution is the fact that the execution with $a = 0$

and $b = 1$ is correct with respect to regularity and incorrect with respect to atomicity. In that execution, and considering the two consecutive read operations $r(a)$ and $r(b)$, the "first", namely $r(a)$, obtains the "new" value ($a = 0$) while the "second", namely $r(b)$, obtains the "old" value ($b = 1$). This is called a *new/old inversion*. This is actually *the exact* difference between regularity and atomicity: an atomic register is a regular register that does not allow for new/old inversions. This important observation is formalized by the following theorem.

Let $\widehat{H}$ be a history of a SWMR regular register. Recall that $\rightarrow_H$ is an irreflexive partial order relation on the set of read and write operations issued during that history: $op \rightarrow_H op'$ if $resp[op]$ occurs before $inv[op']$ (i.e., $resp[op] <_H inv[op']$ in the associated event-based history). Assume without loss of generality that all write operations write distinct values, and let $\pi(r)$ be the write operation that wrote the value obtained by the read operation r (for example, in Figure 1, the first read operation r obtains value 1, and consequently $\pi(r)$ is the first write operation issued by the writer).

Theorem 1 *A SWMR atomic register is a SWMR regular register such that any of its histories $\widehat{H}$ satisfies the following additional property. Let $r1$ and $r2$ be any two read operations:* $(r1 \rightarrow_H r2) \Rightarrow \neg\big(\pi(r2) \rightarrow_H \pi(r1)\big)$.

This theorem states that a SWMR regular register without any new/old inversion is atomic. Looking again at Figure 1, as $r(a) \rightarrow_H r(b)$ and $w(1) \rightarrow_H w(0)$, it is not possible to have $\pi(r(b)) = w(1)$ and $\pi(r(a)) = w(0)$ while ensuring atomicity.

This theorem is very useful to show that an algorithm implements atomic registers. It is used as follows. To prove that an algorithm ensures atomicity, we proceed incrementally proving first that the register is safe, then that it is regular, and finally that it does allow for new/old inversions.

Proof The fact that an atomic register is regular and satisfies the "no new/old inversion" property is an immediate consequence of the definition of atomicity: any history $\widehat{H}$ of an atomic register is equivalent to a legal sequential history $\widehat{S}$ that respects the partial order $\rightarrow_H$ on its operations: two successive read operations that overlap the same write cannot obtain first the new value and then the old value in $\widehat{H}$. Otherwise, no witness sequential history $\widehat{S}$ would be legal, precisely because this history would need to preserve the order of the read operations of $\widehat{H}$.

What is thus left to show is the other direction: namely, a regular register whose executions have no new/old inversion is atomic. Let us first observe that, as we consider a single writer, all write operations are totally ordered (remember that processes are sequential). Let $\stackrel{w}{\rightarrow}$ denote this total order relation. Moreover, let us associate with each write operation w a sequence number $sn(w)$, the first write being numbered 1, etc. For consistency purposes, we also assume that there is an initial write operation before any other operation (or equivalently, the register has an initial value whose sequence number is 1). Let us also associate with each read operation r the sequence number $sn(\pi(r))$, i.e., the sequence number of the write operation $\pi(r)$ that wrote the value read by r.

Let us observe that, due to the fact that the register is regular and there is no new/old inversion, we have the following: for any two read operations $r1$ and $r2$: $(r1 \rightarrow_H r2) \Rightarrow \big(sn(\pi(r1)) \leq sn(\pi(r2))\big)$.

We now show that, for any history $\widehat{H}$, there is a total order $\widehat{S}$ that (a) is equivalent to $\widehat{H}$ (i.e., it includes the same operations), (b) is legal and (c) respects the partial order on the operations defined by $\widehat{H}$. $\widehat{S}$ is built as follows. We start from the total order on the write operations $\stackrel{w}{\rightarrow}$ (that is included in $\widehat{H}$) and insert the read operations as follows:

- A read operation r is inserted just after the associated write operation $\pi(r)$.

- If two read operations $r1$ and $r2$ are such that $sn(r1) = sn(r2)$, then insert first the one that starts first in $\widehat{H}$. (Let us remind that, at the level of the invocation/response events, every two events are totally ordered.)

By its very construction, $\widehat{S}$ includes all the operations of $\widehat{H}$: hence $\widehat{S}$ and $\widehat{H}$ are equivalent. $\widehat{S}$ is trivially a total order (as all the operations are ordered according to the sequence numbers they are associated with). Moreover, this total order is an extension of $\widehat{H}$ as it only adds an order on operations that are concurrent in $\widehat{H}$ (when there is no read operation concurrent with a write operation, there is no difference between regularity and atomicity). Finally, $\widehat{S}$ is legal as each read obtains the last written value that precedes it in this total order. Hence, history $\widehat{H}$ is linearizable. It follows that the register is atomic. $\Box_{Theorem\ 1}$

It follows from the fact that atomicity (linearizability) is a local property that a set of SWMR regular registers behave atomically if each of them -independently from the others- is written by a single process and satisfies the "no new/old inversion" property.

3.5 From very weak to very strong registers

To summarize, there are different kinds of registers and the differences depend on several dimensions. It is appealing to ask whether registers of strong kinds can be constructed in software (emulated) using registers of weak kinds. In fact, and this might look surprising, it is indeed possible to emulate a multi-valued MWMR atomic register using binary SWSR safe registers. Next sections are precisely devoted to showing that.

In general, what we call a (register) *transformation* is here an algorithm that builds a high level register R from low level registers. These low level registers are usually called *base registers*. Of course, for a transformation to be interesting, the base registers it uses have to provide weaker properties than the high level register we want to construct. Typically:

- The base registers are safe (resp., regular) while the high level register is regular (resp., atomic).
- The base registers are SWSR (resp., SWMR) while the constructed register is SWMR (resp., MWMR).
- The base registers are binary whereas the high level register is multi-valued.

The transformations also depend on the number and size of base registers considered. Basically:

- The number of base registers needed to build the high level register might or not depend on the total number of processes in the system, i.e., readers and writers.
- The additional control information used to build the high level register might be bounded or not. Sometimes, the transformation algorithm uses sequence numbers that can arbitrarily grow and is inherently unbounded. In general, and except for few constructions, bounded transformations are much more difficult to design and prove correct than unbounded ones. From a complexity (cost) point of view, bounded ones are always better.

In the following, we proceed as follows.

1. We illustrate the notion of transformation algorithm by presenting first two simple (bounded) algorithms. The first constructs a SWMR safe register out of a number of SWSR safe registers. The second builds a binary SWMR regular register out of a binary SWMR safe register. The combination of these algorithms already shows that we can implement a binary SWMR regular register using a number of binary SWSR safe registers.

2. We then show how to transform a binary SWMR register that ensures some semantics (safe, regular or atomic) into a multi-valued SWMR register that preserves that semantics. The three transformations we present here are all bounded. The combination with the second of these with those above shows that we can implement a multi-valued SWMR regular register using a number of binary SWSR safe registers.

3. We finally show how to go from a SWSR regular register into a MWMR atomic register. We go three intermediate (unbounded) transformations: from a SWSR regular register into a SWSR atomic register, then to a SWMR atomic register, and finally to a MWMR register. These, with the combination pointed out above, shows that we can construct a multi-valued MWMR atomic register using binary SWSR safe registers.

4 Two very simple bounded transformations

This section presents two very simple bounded transformations. We focus on safe and regular registers . (Recall that these kinds of registers are defined for systems where there is a single writer for each such register.) The first transformation extends such a register, being safe or regular, from single reader to multi-readers. The second transformation is used to implement a shared safe bit into a regular one.

4.1 Safe/regular registers: from single reader to multi-reader

We present here an algorithm that implements a SWMR safe (resp., regular) register from SWSR safe (regular) registers. In short, the transformation allows for for multiple readers instead of single readers. Not surprisingly, the idea is to emulate the multi-reader register using several single-reader registers

The transformation, described in Figure 4, is very simple. The constructed register R is built from n SWSR base registers $REG[1 : n]$, one per reader process. (We consider a system of n processes and all are potential readers.) A reader p_i reads the base register $REG[i]$ it is associated with, while the single writer writes all the base registers (in any order).

It is important to see that this transformation is bounded: it uses no additional control information, and each base register can be of the same size (measured in number of bits) as the multi-readers register we want to build.

Interestingly, with the same algorithm, if the base SWSR registers are regular, than the resulting SWMR register we then obtain is regular.

operation $R.write(v)$:
  for_all j **in** $\{1, \dots, n\}$ **do** $REG[i] \leftarrow v$ **end_do**;
  return ()

operation $R.read()$ **issued by** p_i :
  return ($REG[i]$)

Figure 4: From SWSR safe/regular to SWMR safe/regular (bounded transformation)

Theorem 2 *Given one base safe (resp., regular) SWSR register per reader, the algorithm described in Figure 4 constructs a SWMR safe (resp., regular) register.*

Proof Assume first that base registers are safe SWSR registers. It follows directly from the algorithm that a read of R (i.e., $R.read()$) that is not concurrent with a $R.write()$ operation obtains the last value deposited in the register R. The obtained register R is consequently safe while being SWMR.

Let us now consider the case where the base registers are regular. We will argue that the high-level register R constructed by the algorithm is a SWMR regular one. The fact that R allows for multiple readers is by construction. Because a regular register is safe, and by the argument above (for the case where the base registers are safe), we only need to show that a read operation $R.read()$ that is concurrent with one or more write operations $R.write(v)$, $R.write(v')$, etc., returns one of the values $v, v', \ldots$ written by these concurrent write operations, or the value of R before these write operations.

Let p_i be any process that reads some value from R. When p_i reads the base register $REG[i]$, while executing operation $R.read()$, p_i obtains (a) the value of a concurrent write on this base register $REG[i]$ (if any) or (b) the last value written on $REG[i]$ before such concurrent write operations. This is because the base register $REG[i]$ is itself regular. In case (a), the value v obtained is from a $R.write(v)$ that is concurrent with the $R.read()$ of p_i. In case (b), the value v obtained can either be (b.1) from a $R.write(v)$ that is concurrent with the $R.read()$ of p_i , or (b.2) from the last value written by a $R.write()$ before the $R.read()$ of p_i. It follows that the constructed register R is regular. $\square_{Theorem\ 2}$

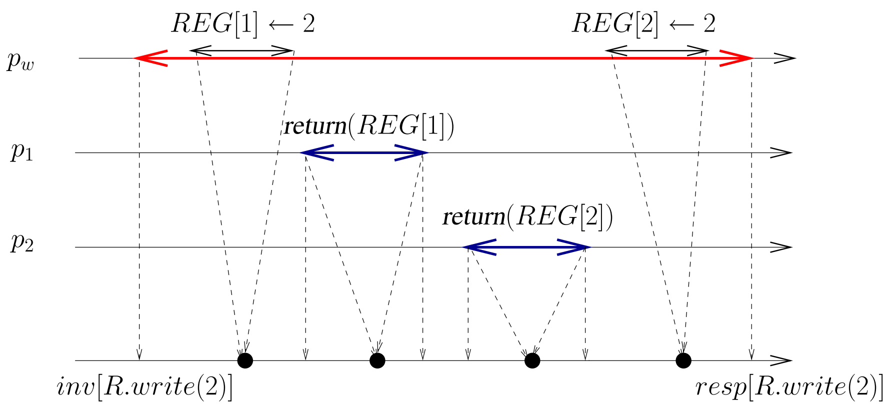

Figure 5: A counter-example

It is important to see that the algorithm of Figure 4 does not build a SWMR atomic register even when every base register $REG[i]$ is a SWSR atomic register. Roughly speaking, this is because the transformation can cause a new/old inversion problem, even if the base registers preclude these. To show this, let us consider the counter-example described in Figure 5. The example involves one writer p_w and two readers p_1 and p_2. Assume the register R obtained from the algorithm contains initially the value 1 (which means that we initially have $REG[1] = REG[2] = 1$). To write the value 2 in R, the writer first executes $REG[1] \leftarrow 2$ and then $REG[2] \leftarrow 2$. The duration between these two write operations on base registers can be arbitrarily long. (Remember that processes are asynchronous and there is no bound on their execution speed). Concurrently, p_1 reads $REG[1]$ and returns 2 while later (as indicated on the figure) p_2 reads $REG[2]$ and returns 1. The linearization order on the two base atomic registers is depicted on the figure (bold dots). It is easy to see that, from the point of view of the constructed register R, there is a new/old inversion as p_1 reads first and obtains the new value, while p_2 reads after p_1 and obtains the old value. The constructed register is consequently not atomic.

4.2 Binary multi-reader registers: from safe to regular

The aim is here to build a regular binary register from a safe binary register, i.e., to construct a regular bit out of a safe one. As we shall see, the algorithm is very simple, precisely because the register to be constructed, R, can only contain one out of two values (0 or 1).

The difference between a safe and a regular register is only visible in the face of concurrency. That is, the value to be returned in the regular case has to be a value concurrently written or the last value written, whereas no such restriction exists for safe semantics. The fact that we only consider a shared bit means however that the issue to be addressed is restricted: only one out of two values can be returned anyway. To illustrate the issue, assume that the regular register is directly implemented using a safe base register: every read (resp. write) on the high-level register is directly translated into a read (resp. write) on the base (safe) register. Assume this register has the value 0 and there is a write operation that writes the very same value 0. As the base register is only safe, it is possible that a concurrent read operation obtains the value 1, which might have never been written.

The way to fix this problem is to preclude the actual writing in the base register if the value to be written in the high-level register is the *same* as the value previously written. If the value to be written is *different* from the previous value, then it is okay to write the value in the case register: a concurrent read can obtain the other value (remember that only two values are possible) and this is fine with the regularity semantics. With this strategy, a read operation concurrent with one or more write operations obtains the value before these write operations or the value written by one of these operations.

The transformation algorithm is described in Figure 6. Besides a safe register REG shared between the reader and the writer, the algorithm requires that the writer uses a local register $prev\_val$ that contains the previous value that has been written in the base safe register REG. Before writing a value v in the high-level regular register, the writer checks if this value v is different from the value in $prev\_val$ and, only in that case, v is written in REG.

operation $R.write(v)$:
  if $(prev\_val \neq v)$ **then** $REG \leftarrow v$;
          $prev\_val \leftarrow v$ **end_if**;
  return ()

operation $R.read()$ **issued by** p_i :
  return (REG)

Figure 6: From binary safe to binary regular register (bounded transformation)

Theorem 3 *Given a SWMR binary safe register, the algorithm described in Figure 6 implements a SWMR binary regular register.*

Proof The proof is an immediate consequence of the following facts. (1) As the underlying base register is safe, a read that is not concurrent with a write obtains the last written value. (2) As the underlying base register always alternate between 0 and 1, a read concurrent with one or more write operations obtains the value of the base register before these write operations or one of the values written by such a write operation. $\square_{Theorem\ 3}$

As we pointed out, the transformation heavily exploits the fact that the constructed register R can only contain one out of two possible values (0 or 1). It does not work for multi-valued registers, nor does it implement an atomic bit register. If the safe base binary register is SWSR, the construction provides a SWSR regular binary register.

5 From binary to b-valued registers

This section presents three transformations from binary registers to multi-valued registers. These are called b-valued registers in the sense that their value domain contains b distinct values; we assume that $b > 2$. Our transformations have three characteristics.

1. Although we assume that the base binary registers (bits) are SWMR registers and we transform these into SRMW b-valued registers, our algorithms can also be used to transform SRSW bits into SWSR b-valued registers.
2. Our transformations preserve the semantics of the base registers in the following sense: if the base bits have semantics X (safe, regular or atomic), then the resulting high-level (b-valued) registers also have semantics X (safe, regular or atomic).
3. The transformations are bounded. There is a bound on the number of base registers used, as well as on the amount of memory needed within each register.

5.1 From safe bits to safe b-valued registers

Overview The first algorithm we present here uses a number of safe bits in order to implement a b-valued register. We assume that b is an integer power of 2, i.e., $b = 2^B$ where B is an integer. It follows that (with a possible pre-encoding if the b distinct values are not the consecutive values from 0 until $b-1$) the binary representation of the b-valued register R we want to build consists of exactly B bits. In a sense, any combination of B bits defines a value that belongs to the domain of R (notice that this would not be true if b was not an integer power of 2).

Using this encoding of the value of R, the algorithm is very simple. It consists in using an array $REG[1:B]$ of SWMR safe bit registers to store the current value of R. Given $\mu_i = REG[i]$, the binary representation of the current value of R is $\mu_1 \ldots \mu_B$. The corresponding transformation algorithm is given in Figure 7.

operation $R.write(v)$:
  let $\mu_1 \ldots \mu_B$ be the binary representation of v;
  for_all j **in** $\{1, \ldots, B\}$ **do** $REG[j] \leftarrow \mu_j$ **end_do**;
  return ()

operation $R.read()$ **issued by** p_i:
  for_all j **in** $\{1, \ldots, B\}$ **do** $\mu_j \leftarrow REG[j]$ **end_do**;
  let v **be** the value whose binary representation is $\mu_1 \ldots \mu_B$;
  return (v)

Figure 7: Safe register: from bits to b-valued register

Cost As $B = \log_2(b)$, the memory cost (space complexity) of the construction is logarithmic with respect to the size of the value domain of the constructed register R. This is an immediate consequence of the binary encoding of the values of the high level register R.

Theorem 4 *Given $b = 2^B$ and B SWMR safe bits, the algorithm described in Figure 7 implements a SWMR b-valued safe register.*

Proof The proof is trivial. A read of R that does not overlap a write of R obtains the binary representation of the last value that has been written into R and is consequently correct. A read of R that overlaps a write of R can obtain any of b possible values whose binary encoding uses B bits. As every possible combination of the B base bit registers represent one of the the b values that R can potentially contain (this is because $b = 2^B$), it follows that a read concurrent with a write operation returns a value that belongs to the domain of R. Consequently, R is a b-valued safe register. $\Box_{Theorem\ 4}$

It is interesting to notice that this algorithm does not build a regular register R even when the base bits are regular. A read that overlaps a write operation that changes the value of R from $0 \ldots 0$ to $1 \ldots 1$ (in binary representation) can return any value, i,e, even one that was never written. The reader (the human, not the process) can check that requiring a specific order according to which the array $REG[1 : B]$ is accessed does not address this issue.

5.2 From regular bits to regular b-valued registers

Overview A way to build a SWMR regular b-valued register R from regular bits consists in employing unary encoding. Considering an array $REG[1 : b]$ of SWMR regular bits, the value $v \in [1..b]$ is represented by 0s in bits 1 through $v - 1$ and 1 in the vth bit.

The algorithm is described in Figure 8. The key idea is to write into the $REG[1 : b]$ array in one direction, and to read it in the opposite direction. To write v, the writer first sets $REG[v]$ to 1, and then "cleans" the array REG. The cleaning consists in setting the bits $REG[v - 1]$ until $REG[1]$ to 0. To read, a reader traverses the array $REG[1 : b]$ starting from its first entry ($REG[1]$) and stopping as soon as it discovers an entry j such that $REG[j] = 1$. It then returns j as the result of the read operation. It is important to see that a read operation starts reading first the "cleaned" part of the array. On the other hand, the writing is performed in the opposite direction, from $v - 1$ until 1.

It is also important to notice that, even when no write operation is in progress, it is possible that several entries of the array be equal to 1. The value represented by the array is then the value v such that $REG[v] = 1$ and for all the entries $1 \le j < v$ we have $REG[j] = 0$. Those entries are then the only meaningful entries. The other entries can be seen as a partial evidence on past values of the constructed register.

The algorithm assumes that the register R has an initial value, say v. The array $REG[1 : b]$ is accordingly initialized, i.e., $REG[j] = 0$ for $1 \le j < v$, $REG[v] = 1$, and $REG[j] = 0$ or 1 for $v < j \le b$.

operation $R.write(v)$:
  $REG[v] \leftarrow 1$;
  for j **from** $v - 1$ **step** -1 **until** 1 **do** $REG[j] \leftarrow 0$ **end_do**;
  return ()

operation $R.read()$ **issued by** p_i:
  $j \leftarrow 1$;
  while $(REG[j] = 0)$ **do** $j \leftarrow j + 1$ **end_do**;
  return (j)

Figure 8: Regular register: from bits to b-valued register

A first observation from the writer algorithm is that, once set to 1, the "last" base register $REG[b]$ keeps that value forever. In a sense, setting this register to 1 makes it useless: the writer never writes in it again, and when it has to read it, a reader might by default consider its value to be 1.

A second observation is that the read algorithm does not write base registers. This means that the algorithm handles any number of readers. Of course, the base registers have to be SWMR if there are several readers (as each reader reads the base registers), and can be SWSR when there a single reader is involved.

Cost The memory cost of the transformation algorithm is b base bits, i.e., it is linear with respect to the size of the value domain of the constructed register R. This is a consequence of the unary encoding of these values[1].

Lemma 1 *Consider the algorithm of Figure 8. Any $R.read()$ or $R.write()$ operation terminates. Moreover, the value v returned by a read belongs to the set $\{1, \ldots, b\}$.*

Proof A $R.write()$ operation trivially terminates (as by definition the **for** loop always terminates). For the termination of the $R.read()$ operation, let us first make the following two observations:

- At least one entry of the array REG is initially equal to 1. Then, it follows from the write algorithm that each time the writer changes the value of a base register $REG[x]$ from 1 to 0, it has previously set to 1 another entry $REG[y]$ such hat $x < y \leq b$.

  Consequently, if the writer updates $REG[x]$ from 1 to 0 while concurrently the reader reads $REG[x]$ and obtains the new value 0, we can conclude that a higher entry of the array has the value 1.

- If, while its previous value is 1, the reader reads $REG[x]$ and concurrently the writer updates $REG[x]$ to the same value 1, the reader obtains the value 1, as the base register is regular [2].

It follows from these observations that a sequential scanning of the array REG (starting at $REG[1]$) necessarily encounters en entry $REG[v]$ whose reading returns 1. As the running index of the **while** loop starts at 1 and is increased by 1 each time the loop body is executed, it follows that the loop always terminates, and the value j it returns is such that $1 \leq j \leq b$. $\Box_{Lemma\ 1}$

Remark The previous lemma relies heavily on the fact that the high-level register R can contain only b distinct values. The lemma would no longer be true if the value domain of R was unbounded. A $R.read()$ operation could then never terminate in the case where the writer continuously writes increasing values. To illustrate that, consider the following scenario. Let $R.write(x)$ be the last write operation terminated before the operation $R.read()$, and assume there is no concurrent write operation $R.write(y)$ such that $y < x$. It is possible that, when it reads $REG[x]$, the reader finds $REG[x] = 0$ because another $R.write(y)$ operation (with $y > x$) updated $REG[x]$ from 1 to 0. Now, when it reads $REG[y]$, the reader finds $REG[y] = 0$ because another $R.write(z)$ operation (with $z > y$) updated $REG[y]$ from 1 and so on. The read can then never terminate.

Theorem 5 *Given b SWMR regular bits, the algorithm described in Figure 8 implements a SWMR b-valued regular register.*

[1] Let B be the number of bits required to obtain a binary representation of a value of R. It is important to see that, as $B = \log_2(b)$, the cost of the construction is exponential with respect to this number of bits.

[2] If the base register was only safe, the reader could obtain the value 0.

Proof Let us first consider a read operation that is not concurrent with any write, and let v the last written value. It follows from the write algorithm that, when $R.write(v)$ terminates, the first entry of the array that equals 1 is $REG[v]$ (i.e., $REG[x] = 0$ for $1 \leq x \leq v - 1$). Because a read traverses the array starting from $REG[1]$, then $REG[2]$, etc., it necessarily reads until $REG[v]$ and returns accordingly the value v.

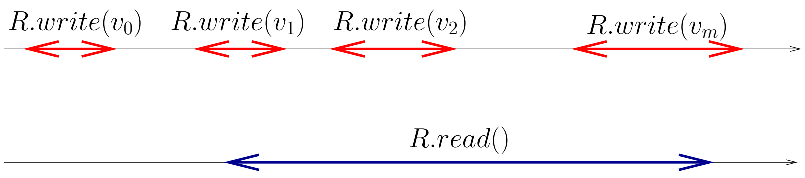

Figure 9: A read with concurrent write operations

Let us now consider a read operation $R.read()$ that is concurrent with one or more write operations $R.write(v_1)$, ..., $R.write(v_m)$ (as depicted in Figure 9). Moreover, let v_0 be the value written by the last write operation that terminated before the operation $R.read()$ starts (or the initial value if there is no such write operation). As a read operation always terminates (Lemma 1), the number of write operations concurrent with the $R.read()$ operation is finite. We have to show that the value v returned by $R.read()$ is one of the values v_0, v_1, ..., v_m. We proceed with a case analysis.

1. $v < v_0$.
   No value that is both smaller than v_0 and different from v_x ($1 \leq x \leq m$) can be output. This is because (1) $R.write(v_0)$ has set to 0 all entries from $v_0 - 1$ until the first one, and only a write of a value v_x can set $REG[v_x]$ to 1; and (2) as the base registers are regular, if $REG[v']$ is updated by a $R.write(v_x)$ operation from 0 to the same value 0, the reader cannot concurrently reads $REG[v'] = 1$. It follows from that observation that, if $R.read()$ returns a value v smaller than v_0, then that value has necessarily been written by a concurrent write operation, and consequently $R.read()$ satisfies the regularity property.

2. $v = v_0$.
   In this case, $R.read()$ trivially satisfies the regularity property. Notice that it is possible that the corresponding write operation be some $R.write(v_x)$ such that $v_x = v_0$.

3. $v > v_0$.
   From $v > v_0$, we can conclude that the read operation obtained 0 when it read $REG[v_0]$. As $REG[v_0]$ was set to 1 by $R.write(v_0)$, this means that there is a $R.write(v')$ operation, issued after $R.write(v_0)$ and concurrent with $R.read()$, such that $v' > v_0$, and that operation has executed $REG[v'] \leftarrow 1$, and has then set to 0 at least all the registers from $REG[v' - 1]$ until $REG[v_0]$. We consider three cases.

   (a) $v_0 < v < v'$.
   In this case, as $REG[v]$ has been set to 0 by $R.write(v')$, we can conclude that there is a $R.write(v)$, issued after $R.write(v')$ and concurrent with $R.read()$, that updated $REG[v]$ from 0 to 1. The value returned by $R.read()$ is consequently a value written by a concurrent write operation. The regularity property is consequently satisfied by $R.read()$.

(b) $v_0 < v = v'$.
The regularity property is then trivially satisfied by $R.read()$, as $R.write(v')$ and $R.read()$ are concurrent.

(c) $v_0 < v' < v$.
In this case, $R.read()$ missed the value 1 in $REG[v']$. This can only be due to a $R.write(v'')$ operation, issued after $R.write(v')$ and concurrent with $R.read()$, such that $v'' > v'$, and that operation has executed $REG[v''] \leftarrow 1$, and has then set to 0 at least all the registers from $REG[v''-1]$ until $REG[v']$.

We are now in the same situation as the one described at the beginning of item 3, where v_0 and $R.write(v')$ are replaced by v' and $R.write(v'')$. As (a) the number of values between v_0 and b is finite and (b) the read operation $R.read()$ terminates, if follows that this operation eventually terminates in 3a or 3b, which completes the proof of the theorem.

$\square_{Theorem\ 5}$

A counter-example for atomicity Figure 10 shows that, even if all base registers are atomic, the algorithm we just presented (Figure 8) does not implement an atomic b-valued register.

Let us assume that $b = 5$ and the initial value of the register R is 3, which means that we initially have $REG[1] = REG[2] = 0$, $REG[3] = 1$ and $REG[4] = REG[5] = 0$. The writer issues first $R.write(1)$ and then $R.write(2)$. There are concurrently two read operations as indicated on the figure. The first read operation returns the value 2 while the second one returns the value 1: there is a new/old inversion. The last line of the figure depicts a linearization order $\widehat{S}$ of the read and write operations on the base binary registers. (As we can see, each base object taken alone is linearizable. This follows from the fact that linearizability is a local property, see the first chapter).

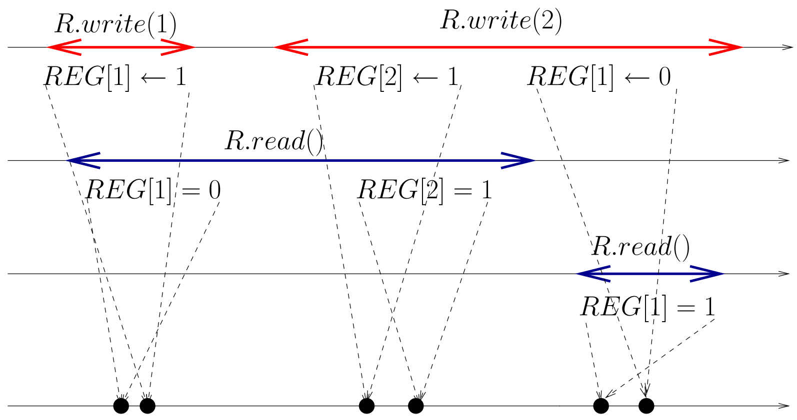

Figure 10: A counter-example for atomicity

5.3 From atomic bits to atomic b-valued registers

As just seen, the previous algorithm does not work if the goal is to build a b-valued atomic register from atomic bits. Interestingly, a relatively simple modification of its read algorithm makes that possible by preventing the new/old inversion phenomenon.

Overview The idea consists in decomposing a $R.read()$ operation in two phases. The first phase is the same as in the previous read algorithm: base registers are read in ascending order, until an entry equal to 1 is found; let j be that entry. The second phase traverses the array in the reverse direction (from j to 1), and determines the smallest entry that contains value 1: this is then returned. So, the returned value is determined by a double scanning of a "meaningful" part of the REG array.

The construction is given in Figure 11. To understand the underlying idea, let us consider the first $R.read()$ operation depicted in Figure 10. After it finds $REG[2] = 1$, the reader changes its scanning direction. The reader then finds $REG[1] = 1$ and returns consequently the value 1. The second read (that starts after the first one) will return the value 1 or 2 according to the value it reads in $REG[1]$. If it reads 1, as in the figure, the read operation returns 1. This shows that, in presence of concurrency, this construction does not strive to eagerly return a value. Instead, the value v returned by a read operation has to be "validated" by an appropriate procedure, namely, all the "preceding" base registers $REG[v-1]$ until $REG[1]$ have to be found equal to 0 when rereading them.

```
operation R.write(v):
    REG[v] ← 1;
    for j from v − 1 step −1 until 1 do REG[j] ← 0 end_do;
    return ()

operation R.read() issued by p_i:
    j_up ← 1;
(1) while (REG[j_up] = 0) do j_up ← j_up + 1 end_do;
(2) j ← j_up;
(3) for j_down from j_up − 1 step −1 until 1 do
(4)         if (REG[j_down] = 1) then j ← j_down end_if
    end_do;
    return (j)
```

Figure 11: Atomic register: from bits to b-valued register

Theorem 6 *Given b SWMR atomic bits, the algorithm described in Figure 11 implements a SWMR atomic b-valued register.*

Proof The proof consists in two parts: (1) we first show that the implemented register is regular, and then (2) we show that it does not allow for new/old inversions. Applying theorem 1 proves then that the constructed register is a SWMR atomic register.

Let us first show that the implemented register is regular. Let $R.read()$ be a read operation and j the value it returns. We consider two cases:

- $j = j\_up$ (j is determined at line 2).
  The value returned is then the same as the one returned by the algorithm described in Figure 8. It follows from theorem 5 that the value read is then either the value of the last preceding write or the new value of an overlapping write.
- $j < j\_up$ (j is determined at line 4; let us observe that, due to the construction, the case $j > j\_up$ cannot happen).

In that case, the read found $REG[j] = 0$ during the ascending loop (line 1), and $REG[j] = 1$ during the descending loop (line 4). Due to the atomicity of the base $REG[j]$ register, this means that a write operation has written $REG[j] = 1$ between these two readings of that base atomic register. It follows that the value j returned has been written by a concurrent write operation.

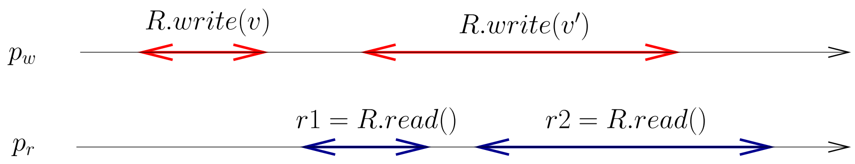

Figure 12: There is no new/old inversion

To show that there is no new/old inversion, let us consider Figure 12. There are two write operations, and two read operations $r1$ and $r2$ that are concurrent with the second write operation. (The fact that the read operations are issued by the same process or different processes is unimportant for the proof.) As the constructed register R is regular, both read operations can return v or v'. If the first read operation $r1$ returns v, the second read can return either v or v' without entailing a new/old inversion. So, let us consider the case where $r1$ returns v'. We show that the second read $r2$ returns v'', where v'' is v' or a value written by a more recent write concurrent with this read. If $v'' = v'$, then there is no new/old inversion. So, let us consider $v'' \neq v'$. As $r1$ returns v', $r1$ has sequentially read $REG[v'] = 1$ and then $REG[v' - 1] = 0$ until $REG[1] = 0$ (lines 2-4). Moreover, $r2$ starts after $r1$ has terminated ($r1 \rightarrow_H r2$ in the associated history $\widehat{H}$).

1. $v'' < v'$. In that case, a write operation has written $REG[v''] = 1$ after $r1$ has read $REG[v''] = 0$ (at line 4) and before $r2$ reads $REG[v''] = 1$ (at line 2 or 4) with $1 \leq v'' < v'$. It follows that this write operation is after $R.write(v')$ (there is a single sequential writer, and $r1$ returns v'). Consequently, $r2$ obtains a value newer than v', hence newer than v: there is no new/old inversion.

2. $v'' > v'$. In that case, $r2$ has read 1 from $REG[v'']$ and then 0 from $REG[v']$ (line 4). As $r1$ terminates (reading $REG[v'] = 1$ and returning v') before $r2$ starts, and write operations are sequential, it follows that there is a write operation, issued after $R.write(v')$, that has updated $REG[v']$ from 1 to 0.

   (a) If that operation is $R.write(v'')$, we conclude that the value v'' read by $r2$ is newer than v', and there is no new/old inversion.

   (b) If that operation is not $R.write(v'')$, it follows that there is another operation $R.write(v''')$, such that $v''' > v'$, that has updated $REG[v']$ from 1 to 0, and that update has been issued after $R.write(v')$ (that set $REG[v']$ to 1), and before $r2$ reads $REG[v'] = 0$.

   Moreover, $R.write(v''')$ is before $R.write(v'')$ (otherwise, the update of $REG[v']$ from 1 to 0 would have been done by $R.write(v'')$).

   It follows that $R.write(v''')$ is after $R.write(v')$ and before $R.write(v'')$, from which we conclude that v'' is newer than v', proving that there is no new/old inversion.

$\square_{Theorem\ 6}$

6 Three (unbounded) atomic register implementations

So far, none of our algorithms implements an atomic register out of a non-atomic register. (Moreover, the one that implements atomic registers implements a b-valued register out of a binary atomic on.) In the following, we present algorithms that implement *unbounded* atomic registers. Such registers can contain any number of distinct values.

We present three transformation algorithms. All use the notion of *sequence number*. In short, this notion represents a concept of logical time. The sequence numbers are associated with each write operation and induce a total order with these operations: the total order is then exploited to ensure atomicity. The use of sequence numbers make these constructions relatively simple. These numbers are written in the base registers, which also means that such registers are unbounded, because the space of sequence numbers is the set of natural numbers.

More specifically, in the algorithms presented in this section, a base register is made up of several fields, namely:

- A data part intended to contain the value v of the constructed high-level register R.
- A control part containing a sequence number and possibly a process identity. The sequence number values increase proportionally with the number of write operations, and is consequently bounded.

Two observations are in order before diving into the details of these transformation algorithms.

1. As we pointed out, the high-level register we construct can be unbounded and might contain, at different points in time, an arbitrary number of distinct values. In fact, the fact that the high-lever register can be unbounded does not mean it has to. This depends on the application that uses this high-level register and the operations that access it.
2. The base registers are unbounded for they contain arbitrarily large sequence numbers. In fact, there are techniques to recycle sequence numbers and bound these constructions. These techniques are however pretty involved and we do not present them here.

6.1 SWSR registers: From unbounded regular to atomic

We show in the following how to implement an SWSR atomic register using a SWSR regular register. The use of sequence numbers make such a construction easy and helps in particular prevent the new/old inversion phenomenon. Preventing this means, by Theorem 1, that the constructed register is atomic.

The construction is described in Figure 13. Exactly one base regular register REG is used in the implementation of the high-level register R. The local variable sn at the writer is used to hold sequence numbers. It is incremented for every new write in R. The scope of the local variable aux used by the reader spans a read operation; it is made up of two fields: a sequence number ($aux.sn$) and a value ($aux.val$).

Each time it writes a value v in the high-level register, R, the writer writes the pair $[sn, v]$ in the base regular register REG. The reader manages two local variables: $last\_sn$ stores the greatest sequence number it has even read in REG, and $last\_val$ stores the corresponding value. When it wants to read R, the reader first reads REG, and then compares $last\_sn$ with the sequence number it has just read in REG. The value with the highest sequence number is the one returned by the reader and this prevents new/old inversions.

Theorem 7 *Given an unbounded SWSR regular register, the algorithm described in Figure 13 constructs a SWSR atomic register.*

```
operation R.write(v):
    sn ← sn + 1;
    REG ← [sn, v];
    return ()

operation R.read():
    aux ← REG;
    if (aux.sn > last_sn) then last_sn ← aux.sn;
                               last_val ← aux.val end_if;
    return (last_val)
```

Figure 13: From regular to atomic: unbounded construction

Proof The proof is similar to the proof of Theorem 1. Let us associate with each read operation r of the high-level register R, the sequence number (denoted $sn(r)$) of the value returned by r: this is possible as the base register is regular and consequently a read always returns a value that has been written with its sequence number, that value being the last written value or a value concurrently written -if any-. Considering an arbitrary history $\widehat{H}$ of register R, we show that $\widehat{H}$ is atomic (linearizable) by building an equivalent sequential history $\widehat{S}$ that is legal and respects the partial order on the operations defined by $\rightarrow_H$.

$\widehat{S}$ is built from the sequence numbers associated with the operations. First, let us order all the write operations according to their sequence numbers. Then, let us order each read operation just after the write operation that has the same sequence number. If two reads operations have the same sequence number, we order first the one whose invocation event is first. (Remember that we consider a SWSR register)

The history $\widehat{S}$ is trivially sequential as all the operations are placed one after the other. Moreover, $\widehat{S}$ is equivalent to $\widehat{H}$ as it is made up of the same operations. $\widehat{S}$ is trivially legal as each read follows the corresponding write operation. We now show that $\widehat{S}$ respects $\rightarrow_H$.

- For any two write operations $w1$ and $w2$ we have either $w1 \rightarrow_H w2$ or $w2 \rightarrow_H w1$. This is because there is a single writer and it is sequential: as the variable sn is increased by 1 between two consecutive write operations, no two write operations have the same sequence number, and these numbers agree on the occurrence order of the write operations. As the total order on the write operations in $\widehat{S}$ is determined by their sequence numbers, it consequently follows their total order in $\widehat{H}$.

- Let $op1$ be a write or a read operation, and $op2$ be a read operation such that $op1 \rightarrow_H op2$. It follows from the algorithm that $sn(op1) \leq sn(op2)$ (where $sn(op)$ is the sequence number of the operation op). The ordering rule guarantees that $op1$ is ordered before $op2$ in $\widehat{S}$.

- Let $op1$ be a read operation, and $op2$ be a write operation. Similarly to the previous item, we then have $sn(op1) < sn(op2)$, and consequently $op1$ is ordered before $op2$ in $\widehat{S}$ (which concludes the proof).

$\Box_{Theorem\ 7}$

One might think of a naive extension of the previous algorithm to construct a SWMR atomic register from base SWSR regular registers. Indeed, we could, at first glance, consider a construction associating one SWSR regular register per reader, and have the writer writes in all of them, each reader reading its dedicated register. Unfortunately, a fast reader might see a new concurrently written value, whereas a reader that comes later sees the old value. This is because the second reader does not know about the sequence number and the value returned by the first reader. The latter stores them locally. In fact, this can happen even if the base SWSR registers are atomic. The construction of a SWMR atomic register from base SWSR atomic registers is addressed in the next section.

6.2 Atomic registers: from unbounded SWSR to SWMR

We presented in Section 4.1 an algorithm that builds a SWMR safe/regular register from similar SWSR base registers. We also pointed out that the corresponding construction does not build a SWMR atomic register even when the base registers are SWSR atomic (see the counter-example presented in Figure 5).

This section provides such a construction: assuming SWSR atomic registers, it shows how to go from single reader registers to a multi-reader register. As it uses sequence numbers, this construction requires unbounded base registers. Thanks to the "comfort" provided by sequence numbers, this construction remains simple, but despite this "comfort", it is not as simple as the previous constructions.

Overview As there are now several possible readers, actually n, we make use of several (n) base SWSR atomic registers: one per reader. The writer writes in all of them. It writes the value as well as a sequence number. The algorithm is depicted in Figure 14.

We prevent new/old inversions (Figure 5) by having the readers "help" each other. The helping is achieved using an array $HELP[1:n, 1:n]$ of SWSR atomic registers. Each register contains a pair (sequence number, value) created and written by the writer in the base registers. More specifically, $HELP[i,j]$ is a SWSR atomic register written only by p_i and read only by p_j. It is used as follows to ensure the atomicity of the high-level SWMR register R that is constructed by the algorithm.

- *Help the others*. Just before returning the value v it has determined (we discuss how this is achieved in the second bullet below), reader p_i helps every other process (reader) p_j by indicating to p_j the last value p_i has read (namely v) and its sequence number sn. This is achieved by having p_i update $HELP[i,j]$ with the pair $[sn, v]$. This prevents p_j from returning in the future a value older than v, i.e., a value whose sequence number would be smaller than sn.
- *Helped by the others*. To determine the value returned by a read operation, a reader p_i first computes the greatest sequence number that it has ever seen in a base register. This computation involves all SWSR atomic registers that p_i can read, i.e., $REG[i]$ and $HELP[j,i]$ for any j. p_i. Reader p_i then returns the value that has the greatest sequence number p_i has computed.

The corresponding construction algorithm is described in Figure 14. Variable aux is a local array used by a reader; its jth entry is used to contain the (sequence number, value) pair that p_j has written in $HELP[j,i]$ in order to help p_i; $aux[j].sn$ and $aux[j].val$ denote the corresponding sequence number and the associated value, respectively. Similarly, reg is a local variable used by a reader p_i to contain the last (sequence number, value) pair that p_i has read from $REG[i]$ ($reg.sn$ and $reg.val$ denote the corresponding fields).

Register $HELP[i,i]$ is used only by p_i, that can consequently keeps its value in a local variable. This means that the SWSR atomic register $HELP[i,i]$ can be used to contain the SWSR atomic register $REG[i]$. It follows that the protocol requires exactly n^2 base SWSR atomic registers.

Theorem 8 *Given n^2 unbounded SWSR atomic registers, the algorithm described in Figure 14 implements a SWMR atomic register.*

Proof As for Theorem 7, the proof consists in showing that the sequence numbers define a linearization of any history $\widehat{H}$.

```
operation R.write(v):
    sn ← sn + 1;
    for_all j in {1, . . . , n} do REG[i] ← [sn, v] end_do;
    return ()

operation R.read() issued by p_i:
    reg ← REG[i];
    for_all j in {1, . . . , n} do aux[j] ← HELP[j, i] end_do;
    let sn_max be max(reg.sn, aux[1].sn, . . . , aux[n].sn);
    let val be reg.val or aux[k].val such that the associated seq number is sn_max;
    for_all j in {1, . . . , n} do HELP[i, j] ← [sn_max, val] end_do;
    return (val)
```

Figure 14: Atomic register: from one reader to multiple readers (unbounded construction)

Considering an history $\widehat{H}$ of the constructed register R, we first build an equivalent sequential history $\widehat{S}$ by ordering all the write operations according to their sequence numbers, and then inserting the read operations as in the proof of Theorem 7. This history is trivially legal as each read operation is ordered just after the write operation that wrote the value that is read. A similar reasoning similar as the one used in Theorem 7, but based on the sequence numbers provided by the arrays $REG[1:n]$ and $HELP[1:n,1:n]$, shows that $\widehat{S}$ respects $\rightarrow_H$. $\Box_{Theorem\ 8}$

6.3 Atomic registers: from unbounded SWMR to MWMR

This section shows how to use sequence numbers to build a MWMR atomic register from n SWMR atomic registers (where n is the number of writers). The construction is simpler than the previous one. An array $REG[1:n]$ of n SWMR atomic registers is used in such a way that p_i is the only process that can write $REG[i]$, while any process can read it. Each register $REG[i]$ stores a (sequence number, value) pair. As before, $X.sn$ and $X.val$ are used to denote the sequence number field and the value field of the register X, respectively. Each $REG[i]$ is initialized to the same pair, namely, $[0, v_0]$ where v_0 is the initial value of R.

The problem to solve consists in allowing the writers to totally order their write operations. To that end, a write operation first computes the highest sequence number that has been used, and defines the next value as the sequence number of its write. Unfortunately, this does not prevent two distinct concurrent write operations to associate the same sequence number with their respective values. A simple way to cope with this problem consists in associating a *times-tamp* with each value, where a times-tamp is a pair made up of a sequence number plus the identity of the process that issues the corresponding write operation.

The timestamping mechanism can be used to define a total order on all the time-stamps as follows. Let $ts1 = [sn1, i]$ and $ts1 = [sn2, j]$ be any two time-stamps. We have:

$$ts1 < ts2 \stackrel{\text{def}}{=} \big((sn1 < sn2) \vee (sn1 = sn2 \wedge i < j)\big).$$

The corresponding construction is described in Figure 15. The meaning of the additional local variables that are used is, we believe, clear from the context.

Theorem 9 *Given n unbounded SWMR atomic registers, the algorithm described in Figure 15 implements a MWMR atomic register.*

operation $R.write(v)$ **issued by** p_i:
  for_all j **in** $\{1, \ldots, n\}$ **do** $reg[j] \leftarrow REG[j]$ **end_do**;
  let sn_max **be** $\max(reg[1].sn, \ldots, reg[n].sn) + 1$;
  $REG[i] \leftarrow [sn\_max, v]$;
  return ()

operation $R.read()$ **issued by** p_i:
  for_all j **in** $\{1, \ldots, n\}$ **do** $reg[j] \leftarrow REG[j]$ **end_do**;
  let k **be** the process identity **such that** $[sn, k]$ is the greatest times-tamp
    among the n time-stamps $[reg[1].sn, 1]$, ... and $[reg[n].sn, n]$;
  return $(reg[k].val)$

Figure 15: Atomic register: from one writer to multiple writers (unbounded construction)

Proof As previously, the proof consists in showing that the timestamps define a linearization of any history $\widehat{H}$.

Considering an history $\widehat{H}$ of the constructed register R, we first build an equivalent sequential history $\widehat{S}$ by ordering all the write operations according to their timestamps, then inserting the read operations as in Theorem 7. This history is trivially legal as each read operation is ordered just after the write operation that wrote the read value. Finally, a reasoning similar to the one used in Theorem 7 but based on timestamps shows that $\widehat{S}$ respects $\rightarrow_H$. $\Box_{Theorem\ 9}$

7 Concluding remark

Section 6 has shown how to build a MWMR atomic register from unbounded SWSR regular registers. From a computability point of view, the price paid by these constructions lies in sequence numbers. The only transformation from safe to regular that has been presented concerns the case of binary registers (Section 4.2). At least three questions are natural to ask:

- How to implement a SWSR atomic bit from a bounded number of SWSR safe bits. This question is addressed in Chapter 4.
- How to implement a SWSR atomic register from a bounded number of SWSR safe bits. This question is addressed in Chapter 5.
- How to implement a MWMR atomic register from bounded SWSR atomic registers. This question is not addressed in this book.

8 Bibliographic notes

The notions of safe, regular and atomic registers have been introduced by Lamport [8]. The theory of wait-free transformation has been investigated by Herlihy [6].

Theorem 1, and the algorithms described in Figure 4, Figure 6, Figure 7 and Figure 8 are due to Lamport [8]. The algorithm described in Figure 11 is due to Vidyasankar [11]. The algorithms described in Figure 14 and 15 are due to Vityani and Awerbuch [15].

The wait-free construction of stronger registers from weaker registers has always been an active research area. The interested reader can consult the following (non-exhaustive!) list where numerous constructions are presented an analyzed [1, 2, 3, 4, 5, 7, 9, 10, 12, 13, 14].

References

[1] Bloom B., Constructing two-writer atomic regsiters. *IEEE Transactions on Computers*, 37:1506-1514, 1988.

[2] Burns J. and Peterson G., Constructing multireaders atomic values from non-atomic values. *Proc. 7th ACM Symposium on Principles of Distributed Computing (PODC'87)*, ACM Press, pp. 222-231, 1987.

[3] Chaudhuri S., Kosa M.J. and Welch J., One-write algorithms for multivalued regular and atomic registers. *Acta Informatica*, 37:161-192, 2000.

[4] Chaudhuri S. and Welch J., Bounds on the cost of multivalued register implementations. *SIAM Journal of Computing*, 2"(2):33(-354, 1994.

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

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

[7] Jayanti P., Burns J. and Peterson G., Almost optimal single reader single writer atomic register. *Journal of Parallel and Distributed Computing*, 60:150-168, 2000.

[8] Lamport. L., On Interprocess Communication, Part 1: Basic formalism, Part II: Algorithms. *Distributed Computing*, 1(2):77-101, 1986.

[9] Li M., Tromp J. and Vityani P., How to share concurrent wait-free variables. *Journal of the ACM*, 43(4):723-746, 1996.

[10] Singh A.K., Anderson J.H. and Gouda M., The elusive atomic register. *Journal of the ACM*, 41(2):331-334, 1994.

[11] Vidyasankar K., Converting Lamport's Regular Register to Atomic Register. *Information Processing Letters*, 28(6):287-290, 1988.

[12] Vidyasankar K., An elegant 1-writer multireader multivalued atomic register. *Information Processing Letters*, 30(5):221-223, 1989.

[13] Vidyasankar K., A very simple cosntruction of 1-writer multireader multivalued atomic variable. *Information Processing Letters*, 37:323-326, 1991.

[14] Vityani P., Simple wait-free multireader registers. *Proc. 16th Int'l Symposium on Distributed Computing (DISC'02)*, Springer-Verlag LNCS #2508, pp. 118-132, 2002.

[15] Vityani P. and Awerbuch B., Atomic shared register access by asynchronous hardware. *Proc. 27th IEEE Symposium on Foundations of Computer Science (FOCS'87)*, IEEE Computer Press, 223-243, 1986.