Fault-tolerant simulation of read/write objects

.

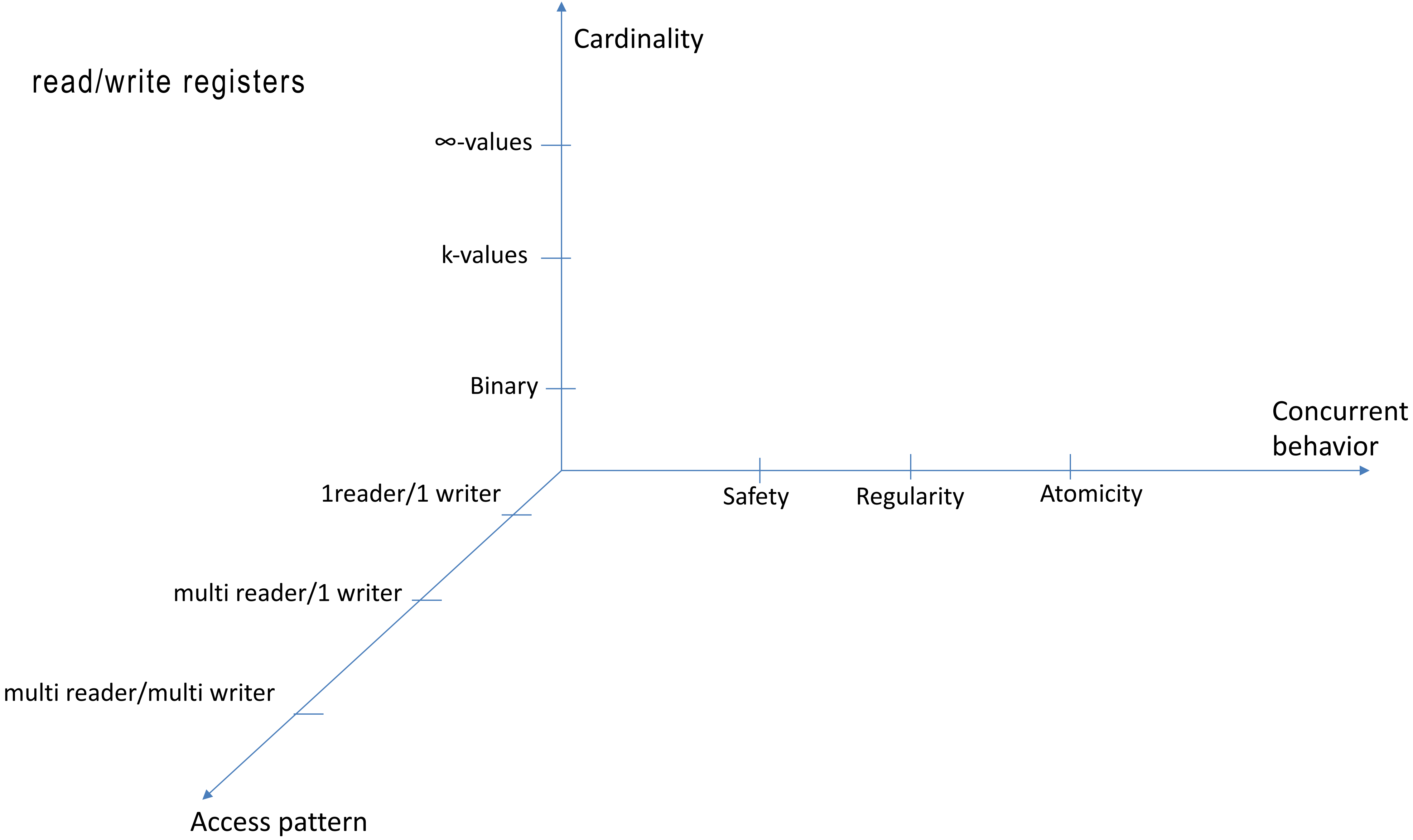
Emmanuelle Anceaume

emmanuelle.anceaume@irisa.fr

Fault tolerant simulations of read/write objects

- In lesson 1, we have seen how we can build high level shared objects by using low level ones:
  - By relying on mutual exclusion section, where objects are updated in critical sections
  - Unfortunately those constructions are very sensitive to delays or failures
  - If some process blocks in the CS, then it blocks all the other processes which are waiting for the CS

read/write registers



Fault tolerant simulations of read/write objects

Safety

- A read that is not concurrent with a write returns the last written value
    - This is the only property ensured by a safe register
    - Thus a safe register does not any guarantee if accessed concurrently: such a register supports only a single write
- If this writer is concurrent with a read, this read can return any value in the range domain of the register

- A binary safe register is thus a bit flickering under concurrency.

Fault tolerant simulations of read/write objects

Regularity

- A regular register ensures, together with the safety property above, that a read that is concurrent with a write returns the value written by that write or the value written by the last preceding write.
- A regular register also only supports a single writer.

- It is important to notice that such a register can, if two consecutive (non-overlapping) reads are concurrent with a write, returns the value being written (the new value) and then returns later the previous value written (the old value). This situation is called the new/old inversion.

Fault tolerant simulations of read/write objects

Atomicity

- An atomic (linearizable) register is one that ensures linearizability.
- Such a register ensures, in addition to the safety and regularity properties above, that a new/old inversion never happens.
  - The second read must return the same or a “newer” value

Proving the properties of registers

- Proving that a register is safe consists only in showing that it respects it sequential specification in absence of concurrency
- Proving that a register is regular or atomic is more difficult

- We introduce the notion of read function f
- The read function is associated to an history and maps for any read operation the write operation that wrote the value returned by the read operation

Proving the properties of registers

We say that a reading function associated with a history H is regular if it satisfies the following two properties:

A1 : $\forall$ r: $\neg(r \rightarrow_H f(r))$ (No read returns a value not written yet)

A2 : $\forall$ r in H: $(w \rightarrow_H r) \Rightarrow (f(r) = w \lor w \rightarrow_H f(r))$ (No read returns a value overwritten)

We say that a reading function is atomic if besides being regular it satisfies the following property:

A3 : $\forall$ r1, r2: $(r1 \rightarrow_H r2) \Rightarrow (f(r1) = f(r2) \lor f(r1) \rightarrow_H f(r2))$ (No new/old inversion)

Constructing atomic registers from safe ones

Theorem:

A multivalued MWMR atomic register can be wait-free implemented with binary SRSW safe registers

Wait-free: any processor that is ready to write() or read() must do it without waiting for the other processors.

Constructing atomic registers from safe ones

Assumptions:

- n processors, p_1,…p_i, p_j, …p_n

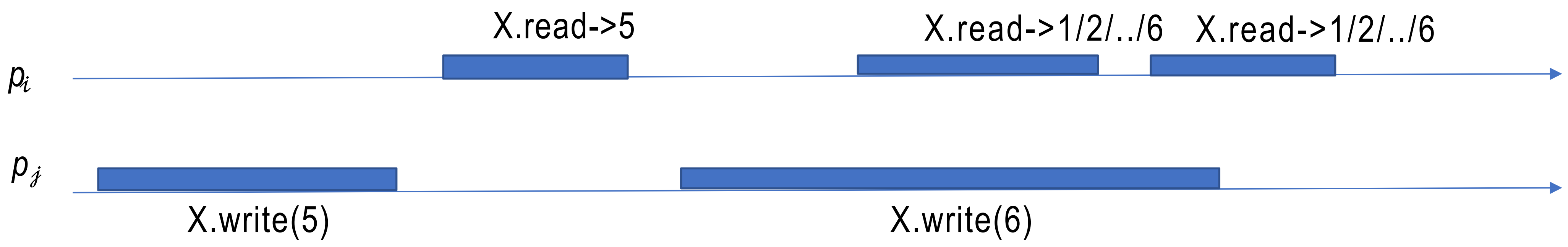
Notations:

- the operations of the base registers: read() and write()
- the operations to be implemented: Read() and Write()

Read/Write safe register

Properties:

- a read() not concurrent with any write() obtains the correct value, i.e., the most recently written one
- a read() that overlaps a write() returns any possible values of the register



[1,..6]-valued safe register X

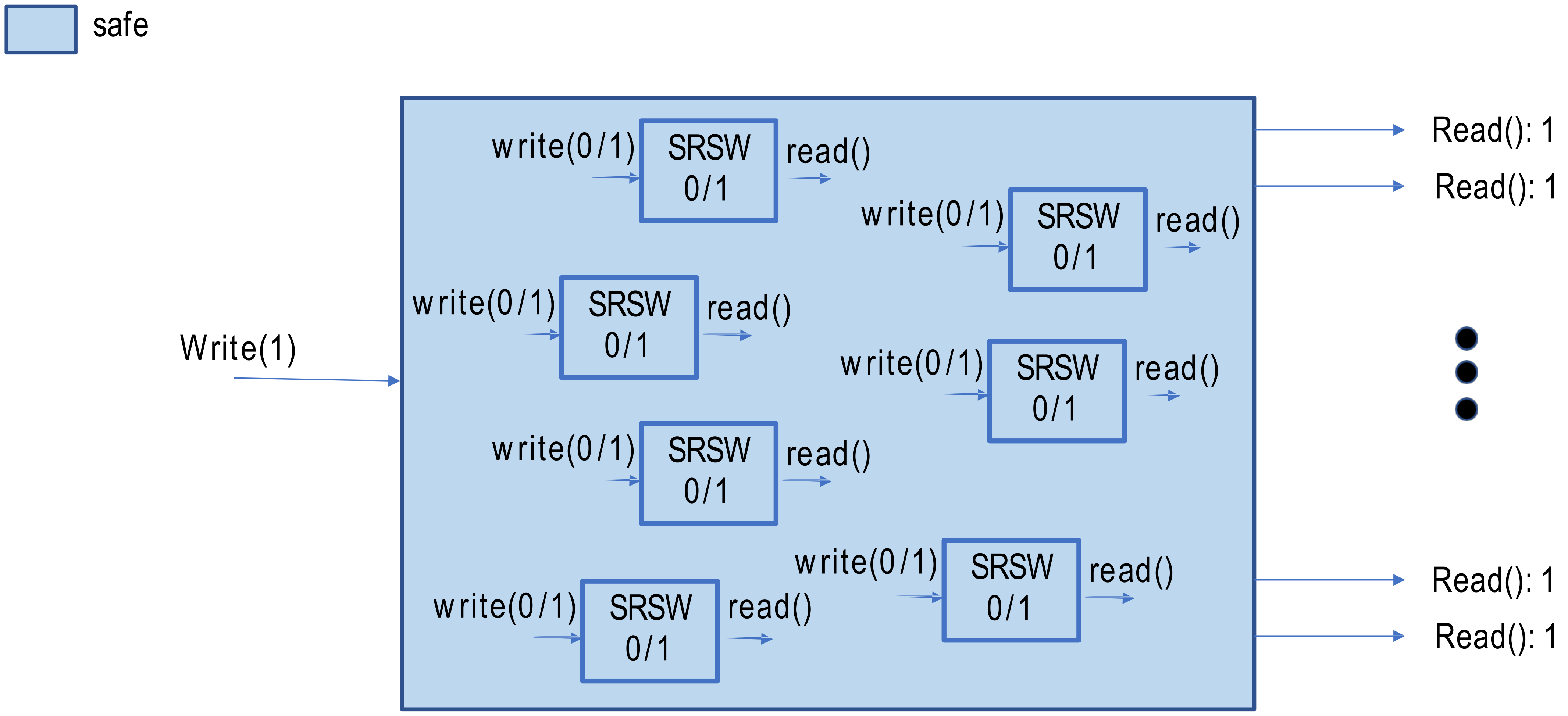
From one reader to multiple readers registers

The following two constructions present constructions that change the number of readers of the registers:

- From 1W1R binary safe register to 1WMR safe binary register
- From 1WMR binary safe to 1WMR binary regular

-> 1W1R binary safe register to 1WMR binary regular

Construction 1:
binary MRSW safe from binary SRSW safe registers

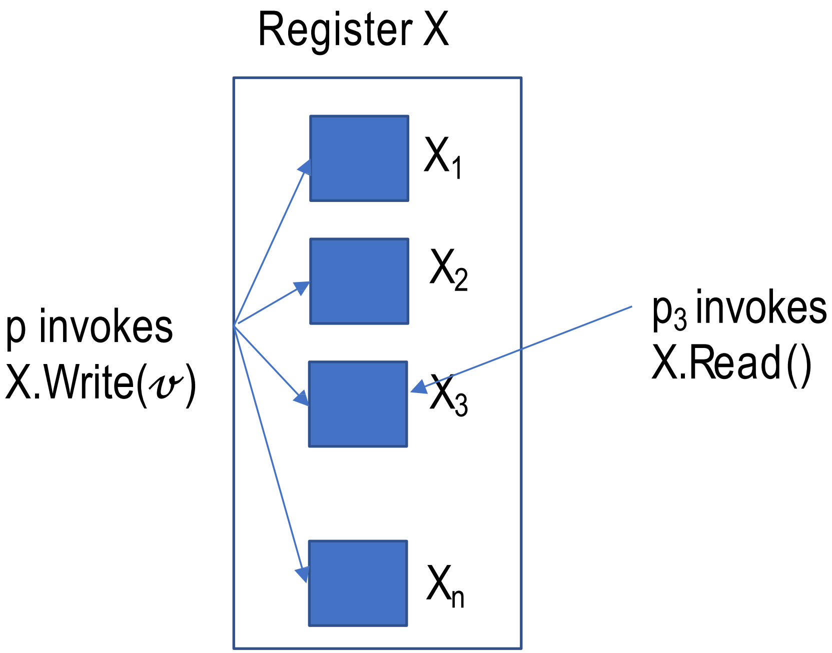


Construction 1:
binary SWMR safe from binary SWSR safe registers

X: binary SWMR safe register we want to build

The writer maintains a copy of the register for each reader Let X_1, ...X_n be *n* binary SWSR safe registers

```
When p invokes X.Write(v):
    for all i in {1, .., n} do
        Xi.write(v)
    return()

When pi invokes X.Read():
   return (Xi.read())
```



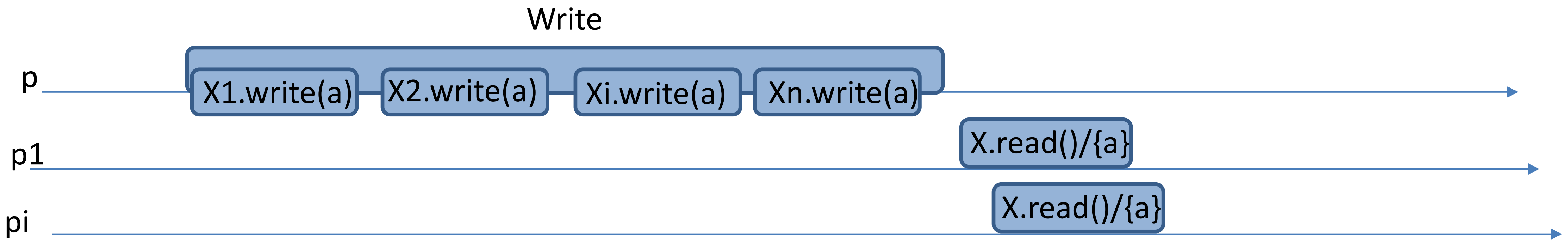
Construction 1:
binary SWMR safe from binary SWSR safe registers

If the X_i are safe registers, then X is a safe register .

Why ?

- Any Read() by p_i that does not overlap a Write() does not overlap a write() of X_i
- Thus if X_i is safe, then this Read() gets the correct value, which shows that X is safe

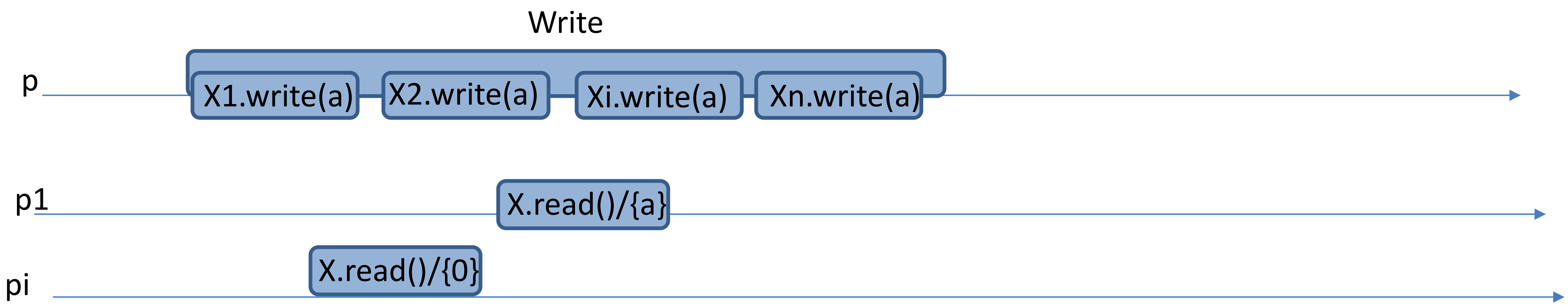
- Note that each safe register has the same size (number of bits) as the register we want to build



Construction 1:

Construction 1 works to build a 1WMR regular register from 1W1R regular registers

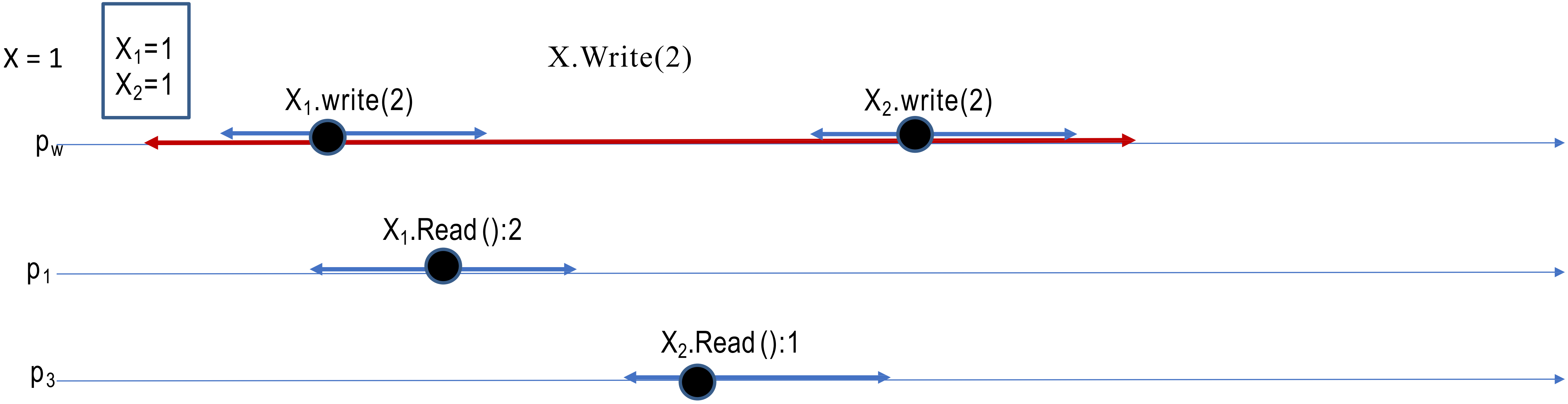
Indeed since regulars registers are also safe, we just need to show that a Read() operation that is concurrent with one or more Write operations returns a concurrently written value or the last written value



Construction 1

Construction 1 preserves safety and regularity but not atomicity.

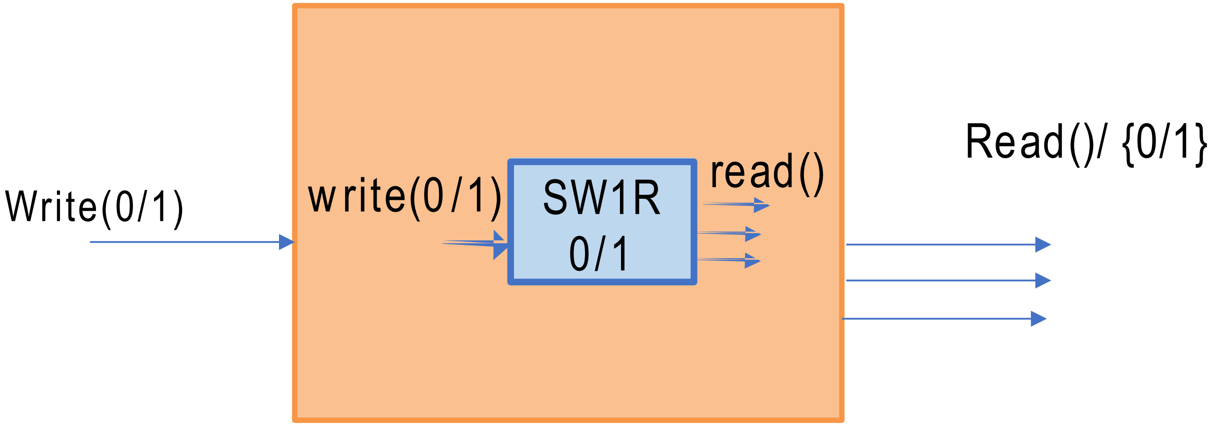
This is because of new/old inversions



Construction 2:
binary SWMR regular from binary SWMR safe registers

safe

regular



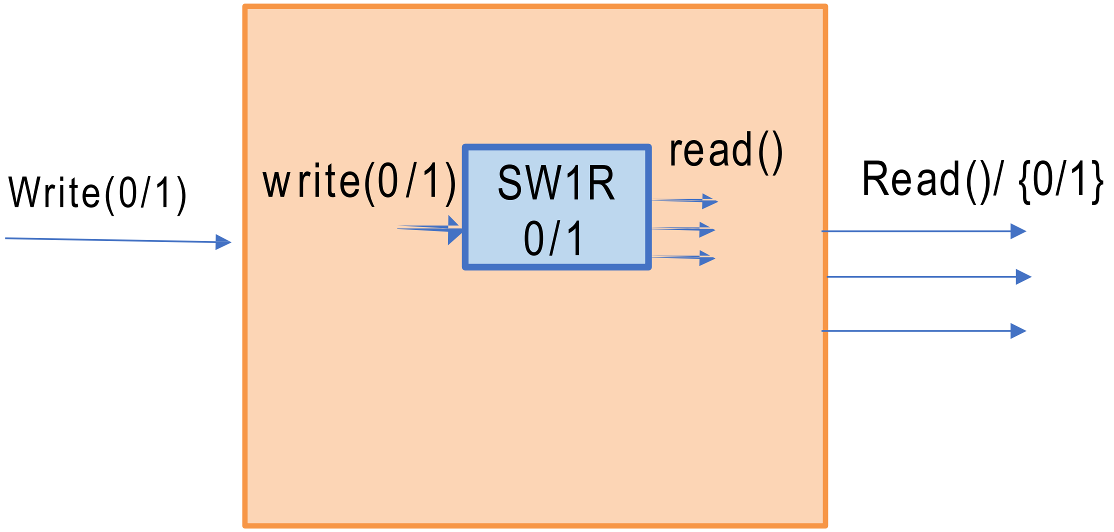
Construction 2:
binary SWSR regular from binary SWMR safe registers

X: binary SWSR regular register we want to build

From a safe SWSR safe register X1

This construction deeply relies on the fact that registers are binary

Recall that a binary safe register can return either {0} or {1} in presence of concurrent writes, even if for instance {1} overwrites {1}



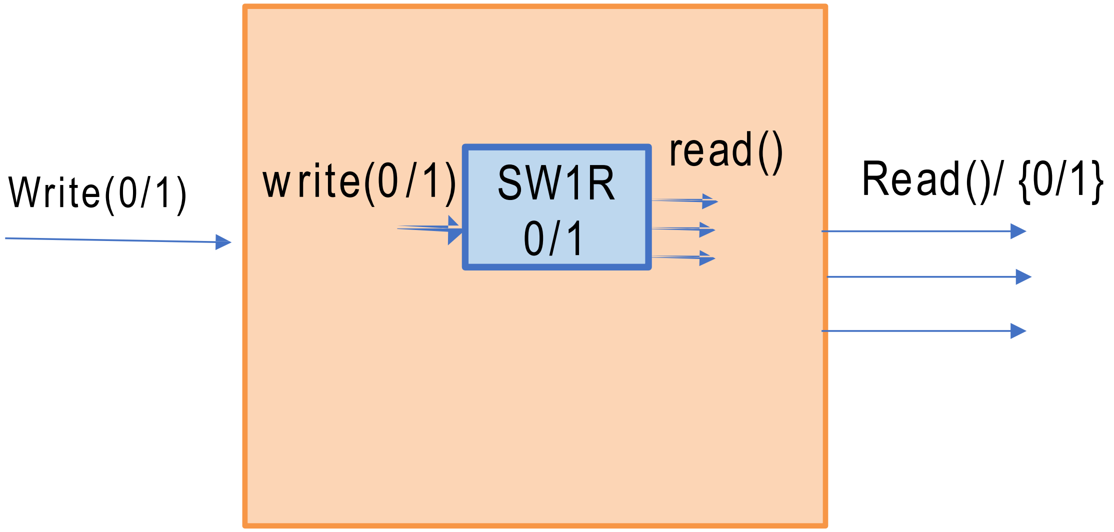
Construction 2:
binary SWMR regular from binary SWMR safe registers

X: binary SWSR regular register we want to build

From a safe SWSR safe register X1

When p invokes X.Write(v):
    if (prev_val $\neq$ v) then
        X1.write(v)
    return()

When pi invokes X.Read():
    return (X1.read())



Construction 2:
binary SWMR regular from binary SWMR safe registers

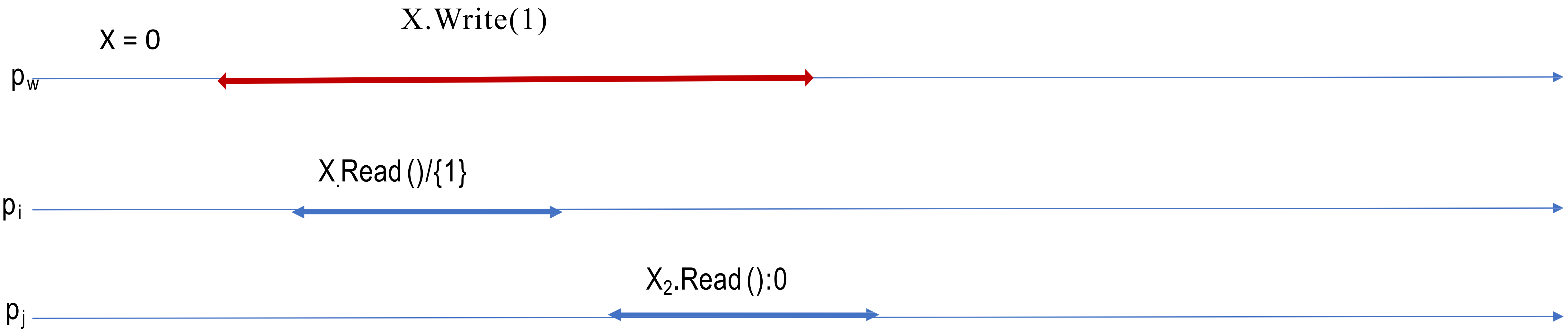
This construction implements a regular 1WMR register from a safe one. Why ?

- By assumption the underlying base register is safe: a read executed in presence of no overlapping operation returns the last written value.
- In presence of concurrent write and read operations, the read operation
  - Either returns the previously written value (if both write operations write the same value)
  - Or the concurrent one or the last previously written one (if both write operation write different values)

Construction 2:
binary SWMR regular from binary SWMR safe registers

Construction 2 does not implement an atomic register from a safe one

This is because of new/old inversions

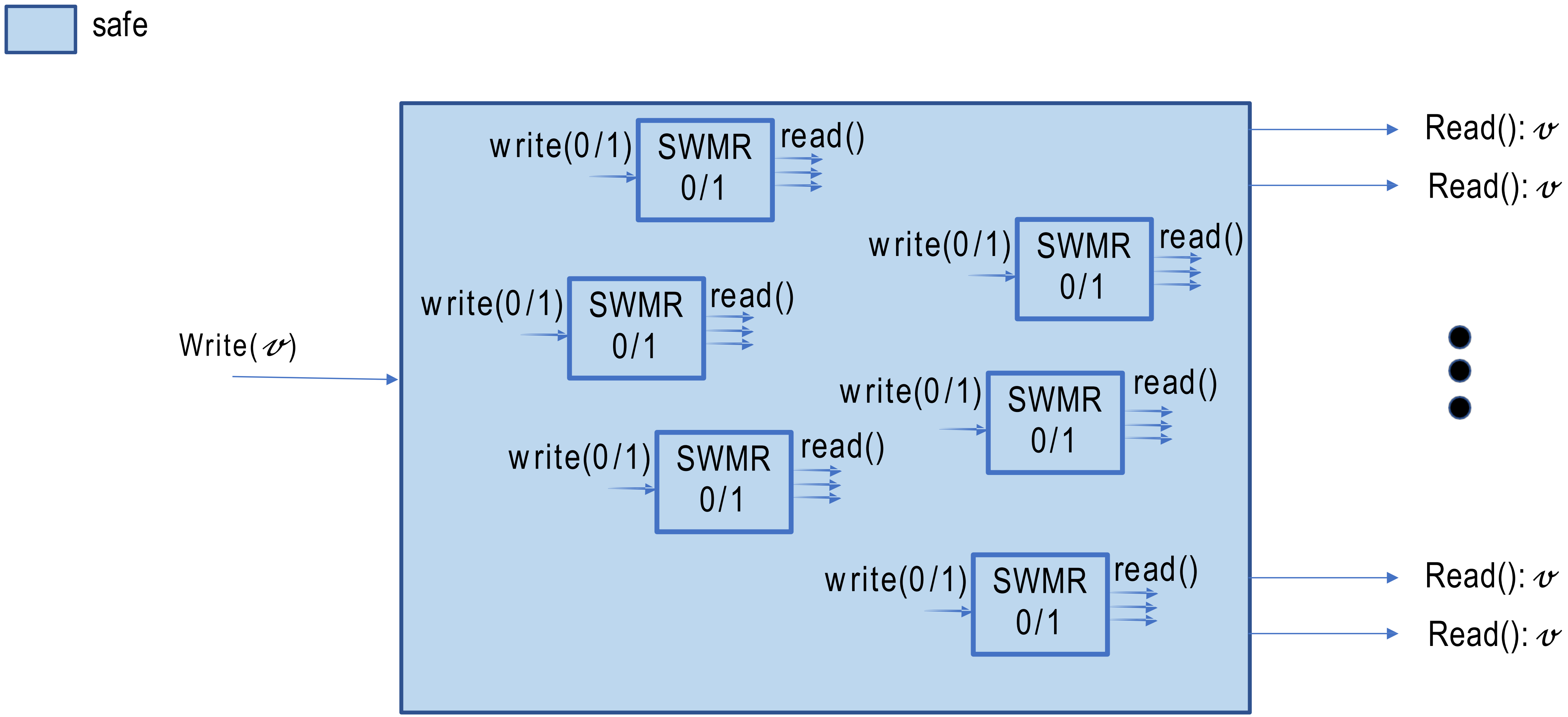


From binary to b-valued registers constructions

The following three constructions present constructions that change the cardinality of the registers:

- From binary safe to b-valued safe
- From binary regular to b-valued regular
- From binary atomic to b-valued atomic

Construction 3:
b-valued SWMR safe from binary SWMR safe registers



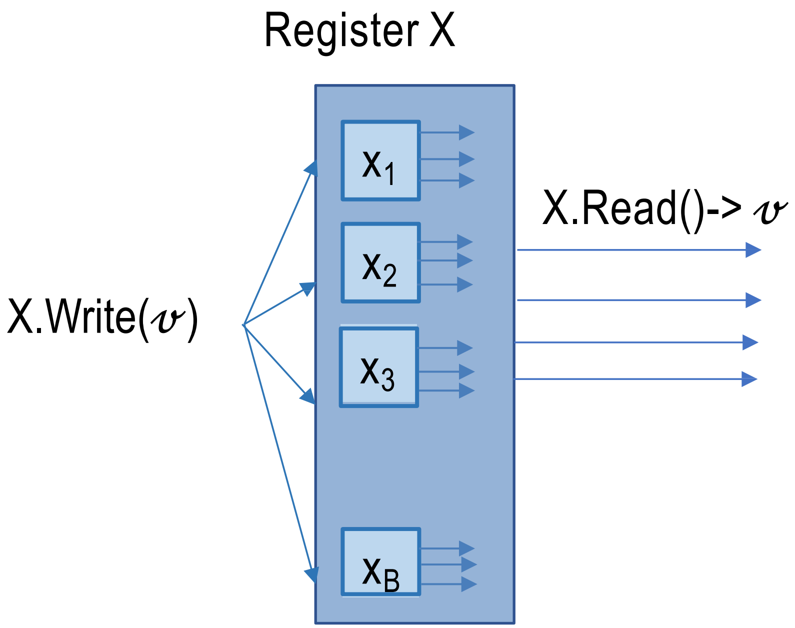
Construction 3:
b-valued SWMR safe from binary SWMR safe registers

X: b-valued-MRSW safe register we want to build

Let B s.t. $b = 2^B$

Let $X_1, \ldots X_B$ be *B* binary MRSW safe registers

```
when p invokes X.Write(v):
    let v1v2....vB be the binary representation of v
        for each i in {1, .., B}
            Xi.write(vi)
    return()

when pi invokes X.Read():
        for each i in {1, .., B} do vi = Xi.read()
        Let v be the value of v1v2....vB
        return (v)
```



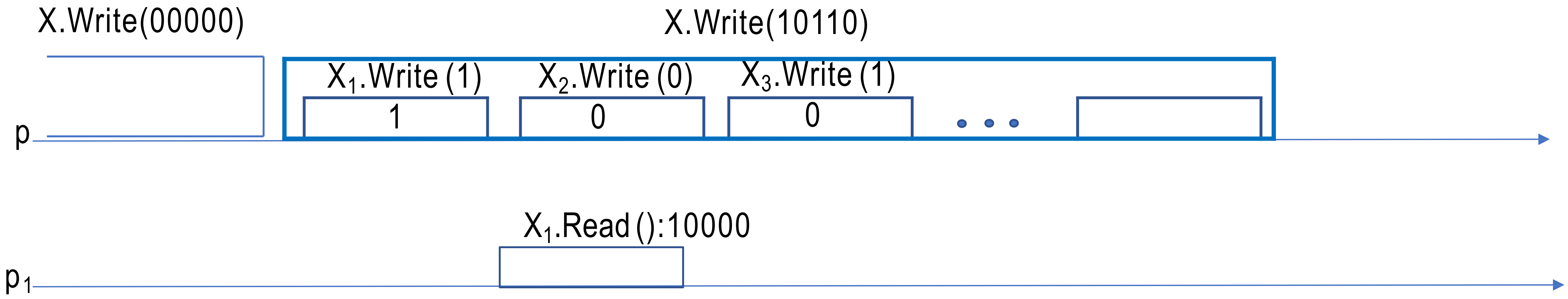
Construction 3:
b-valued SWMR safe from binary SWMR safe registers

If the X_i are binary safe registers, then Construction 3 implements a 2^B safe register

Why ?

- Any Read() that does not overlap a X.Write() returns the value of the binary representation of the last value written.
- A read of X that overlaps a write of X can return any possible value whose binary encoding uses B bits

Construction 3:
b-valued SWMR safe from binary SWMR safe registers

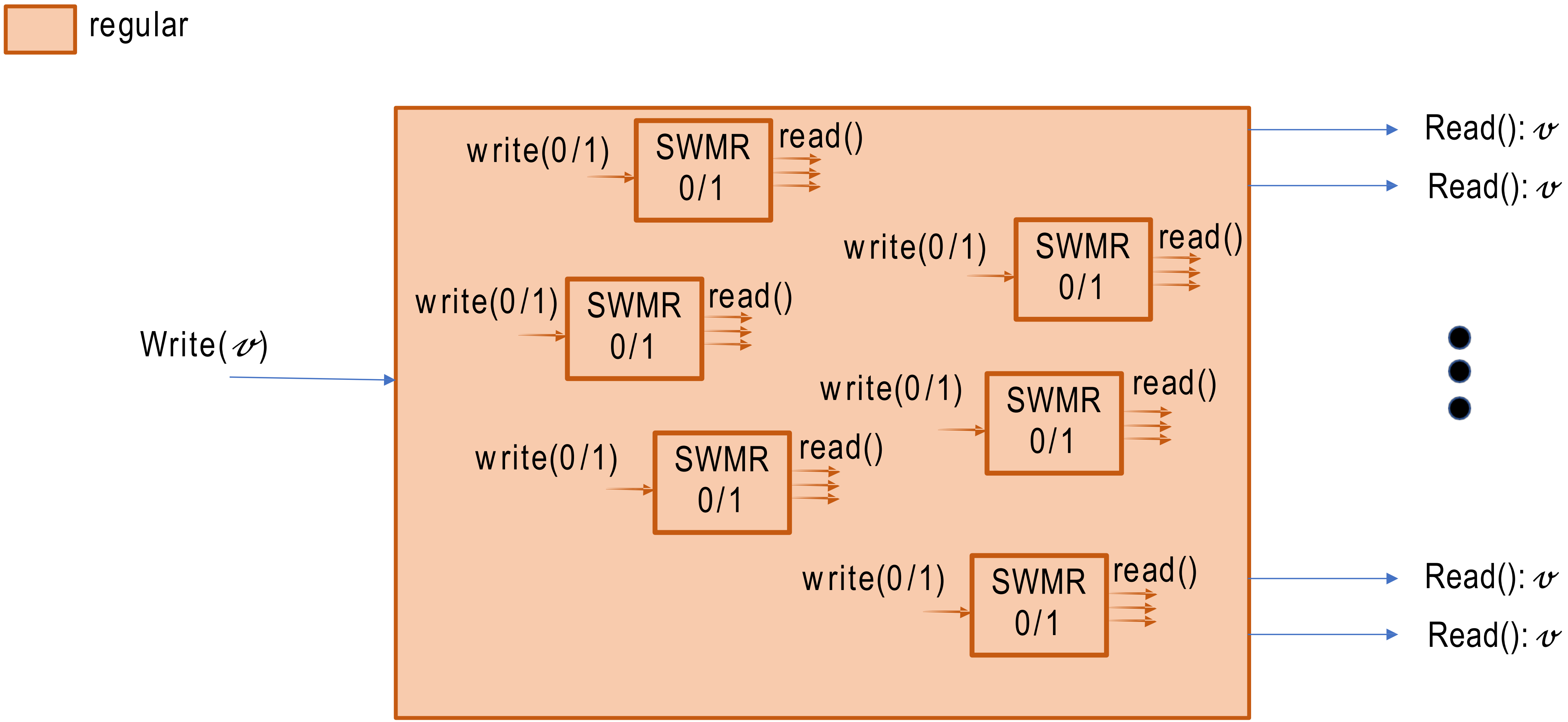


Any value between 10000 and 10100 can be returned.

Construction 3 cannot implement a b-valued SWMR regular register even if the B Boolean registers are regular

Construction 3 cannot implement a b-valued SWMR atomic register even if the B Boolean registers are atomic

Construction 4:
b-valued SWMR regular from binary SWMR regular registers



Construction 4:
b-valued SWMR regular from binary SWMR regular registers

- The construction employs unary-encoding:

Value v in [0,b] is represented by « 0 » in bits 0 through v-1 and « 1 » in bit v

- The construction uses b binary SWMR regular registers to code b distinct values (recall that it was logarithmic in Construction 3)

- The idea is to write in one direction and to read in the opposite direction

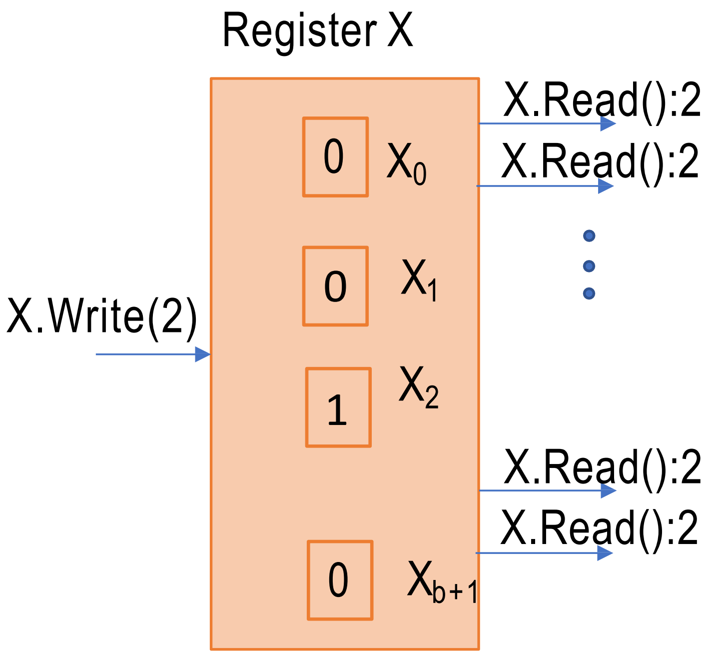
- To write v, the writer firsts sets X_v to “1” and then sets all the other (v-1) registers to “0”
- To read(), the reader starts reading X_0, X_1,… and stops once it founds a register i set to “1” the returned value is i

Construction 4:
b-valued SWMR regular from binary SWMR regular registers

X: b-valued SWMR regular register we want to build

Let $X_0, \ldots X_b$ be $b+1$ binary SWMR regular registers initialized with "0" except one with "1"

```
when p invokes X.Write(v):
    X_v.write(1)
    for each i in {v-1, .., 0}
        X_i.write(0)

when p_i invokes X.Read():
    i:=0
    while (X_i.Read() ≠ 1) do
        i:=i+1
    return (i)
```



Construction 4:
b-valued SWMR regular from binary SWMR regular registers

Q: Does the while stops ?

Yes: a “0” is written only if a “1” is written to its right

Q: Is it true that when a reader reads a “1” then this “1” has been written either by a concurrent write() or by the preceding write()?

Yes: base registers are regular

Note that several 1 can « co-exist » even if there are no concurrent operations. The smallest one refers to the last written value

```
when p invokes X.Write(v):
    X_v.write(1)
    for each i in {v-1, .., 0}
        X_i.write(0)

when p_i invokes X.Read():
    i:=0
    while (X_i.Read() ≠ 1) do
        i:=i+1
    return (i)
```

Construction 4:
b-valued SWMR regular from binary SWMR regular registers

Construction 4 is wait-free

Every X.write(v) operation terminates in a finite number of steps:
the loop only goes through v iterations.

Consider a X:read()

- Remember first that there is initially at least one a valid value v_0 and hence initially a « 1 » in the register.
- Now observe that when the writer changes X_i from 1 to 0, the writer has already set to « 1 » another X_j such that i < j.
- Hence the loop eventually terminates in a finite number of steps.

```
when p invokes X.Write(v):
    X_v.write(1)
    for each i in {v-1, .., 0}
        X_i.write(0)

when p_i invokes X.Read():
    i:=0
    while (X_i.Read() ≠ 1) do
        i:=i+1
    return (i)
```

Construction 4:
b-valued SWMR regular from binary SWMR regular registers

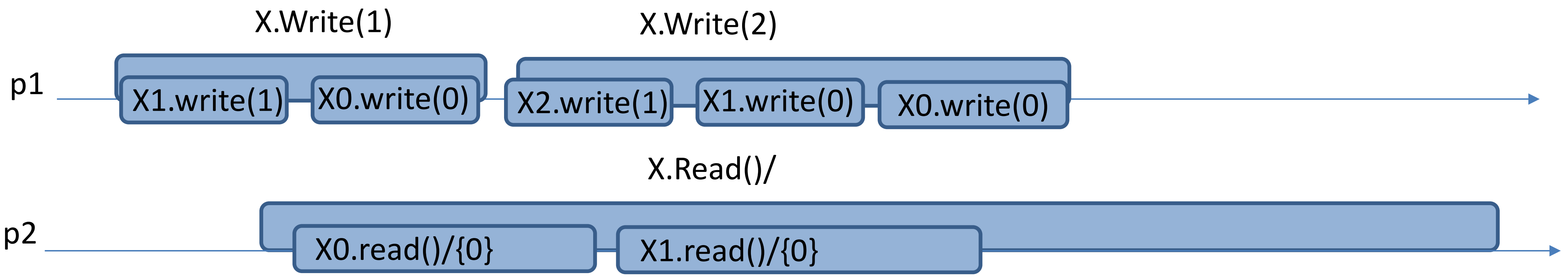
Construction 4 is wait-free

- Note that the previous argument is true because the register can contain up to b distinct values.
- If the range of X was unbounded a read() operation could never terminate if the writer was continously updating the register. Why ?

```
when p invokes X.Write(v):
    X_v.write(1)
    for each i in {v-1, .., 0}
        X_i.write(0)

when p_i invokes X.Read():
    i:=0
    while (X_i.Read() ≠ 1) do
        i:=i+1
    return (i)
```

Construction 4:
b-valued SWMR regular from binary SWMR regular registers

Scenario showing that if the range of values of the register is unbounded the loop of the read does not terminate in a finite number of steps



Construction 4:
b-valued SWMR regular from binary SWMR regular registers

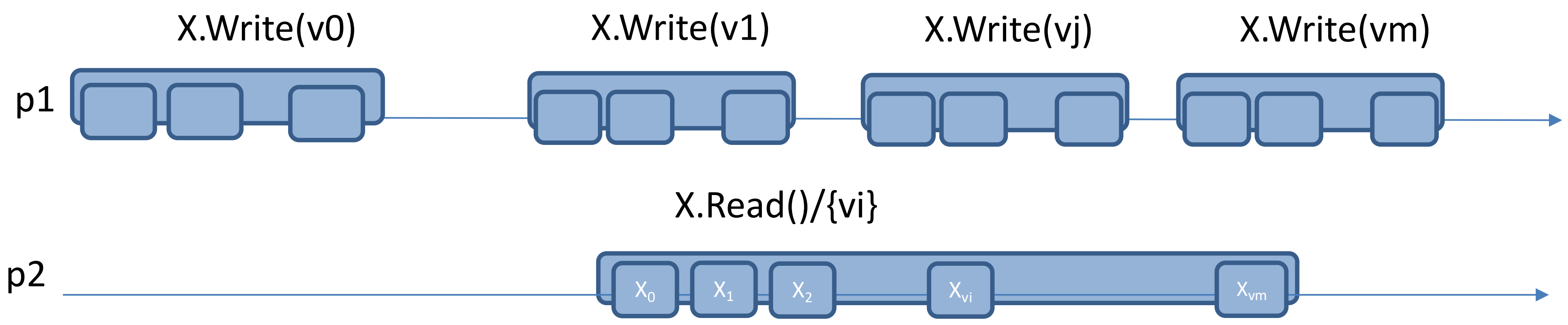
Construction 4 implements a b-valued regular register. Why ?

1. Consider a Read operation that does not overlap any Write(). Let v be the last written value. So the 1rst register equal to « 1 » is X_v, and registers X_0, … X_{v-1} are set to « 0 ». So the the read will return v
2. Consider a Read operation concurrent with Write operations X.write(v_1),…., X.Write(v_m).
   The # of Write operations is bounded because Read() operations terminate
   Let v0 be the last written value preceding X.Read().

```
when p invokes X.Write(v):
    X_v.write(1)
    for each i in {v-1, .., 0}
        X_i.write(0)

when p_i invokes X.Read():
    i:=0
    while (X_i.Read() ≠ 1) do
        i:=i+1
    return (i)
```

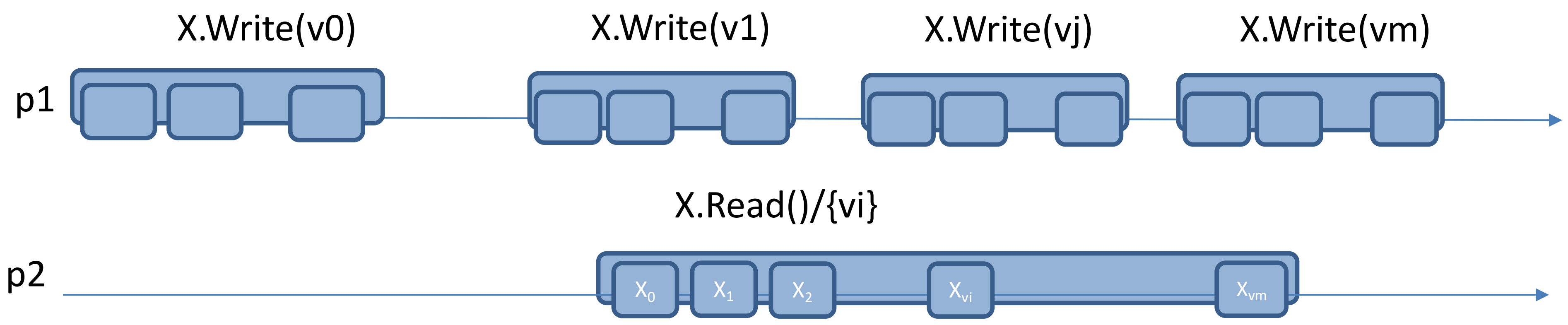
Construction 4:
b-valued SWMR regular from binary SWMR regular registers

We are going to show by induction that each of the basic read() operations on the X_i registers return a value previously written (i.e. v_0) or concurrently written (i.e., v_1,..v_m)



Construction 4:
b-valued SWMR regular from binary SWMR regular registers

Base of the induction:
Since X.Write(v0) sets X_{v0} to “1” and all the “smallest registers to “0”, then X0.read() returns either the value written by the last preceding written value (i.e. “0” if v0>0 or ”1” if v0=1) or the value written by a concurrent Write operation
(recall that base registers are regular)

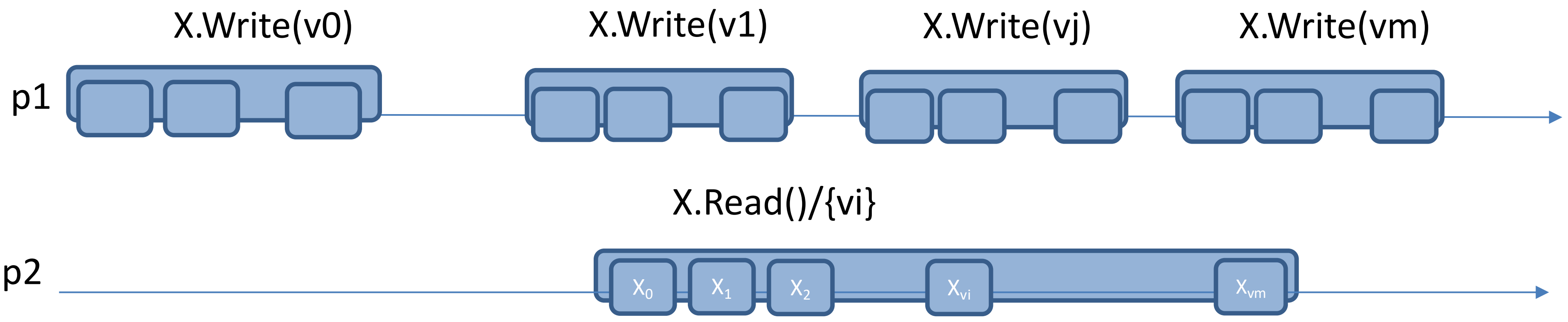


Construction 4:
b-valued SWMR regular from binary SWMR regular registers

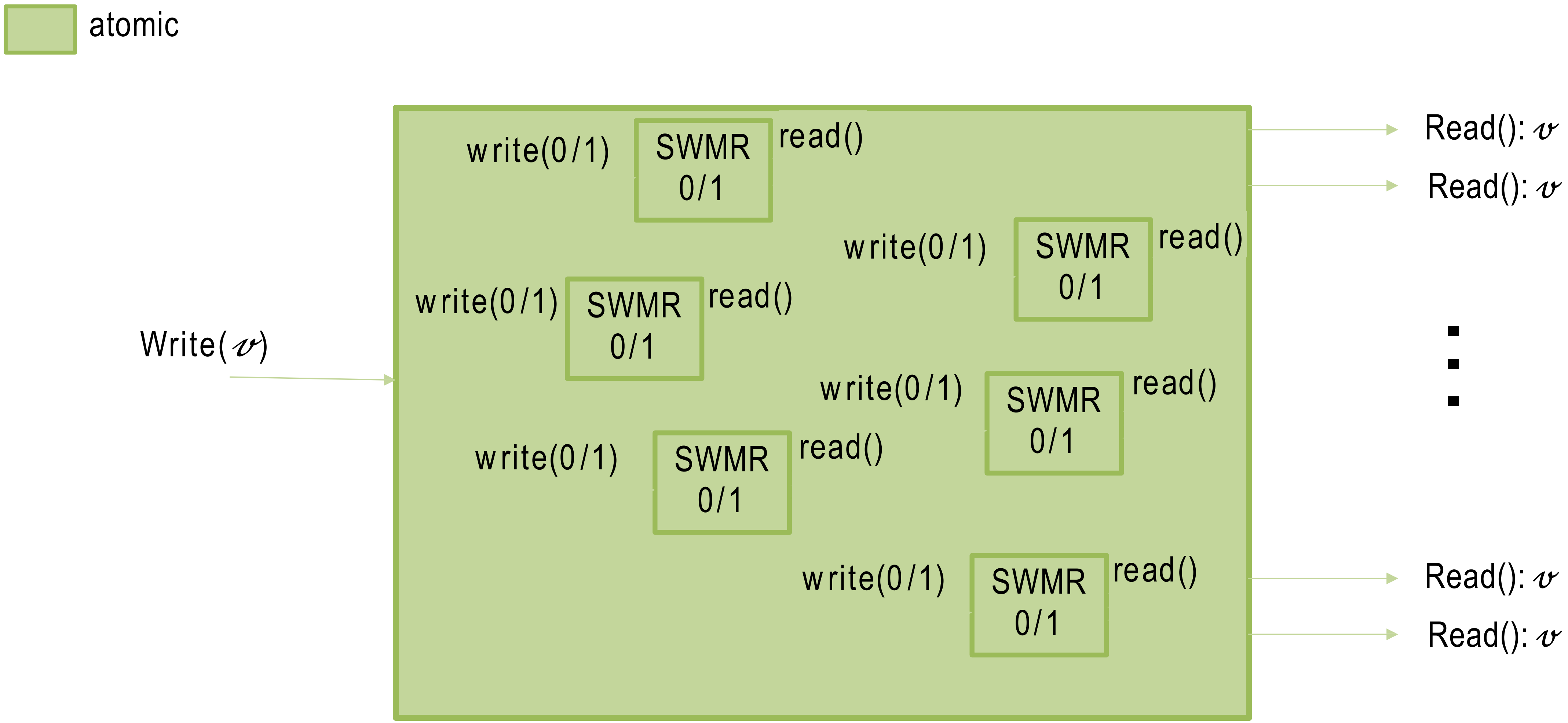
Suppose that the value read in X_j = 0 is the value written by the last preceding Write operation (i.e. X.Write(v0)) or a concurrent one X.Write(vk)).

We must have k > j otherwise X.Write(vk) would not touched Xj.

By the algorithm X.Write(vk) has previously set X_{vk}:=1, X_{vk-1}:=0, …, X_{j+1}:=0. Thus since the base registers are regular, the subsequent read of X_{j+1} performed within the X.Read() can only return the value written by X.Write(vk)) or a subsequent X.Write(vℓ) operation concurrent with X.read()



Construction 5:
b-valued SWMR atomic from binary SWMR atomic registers



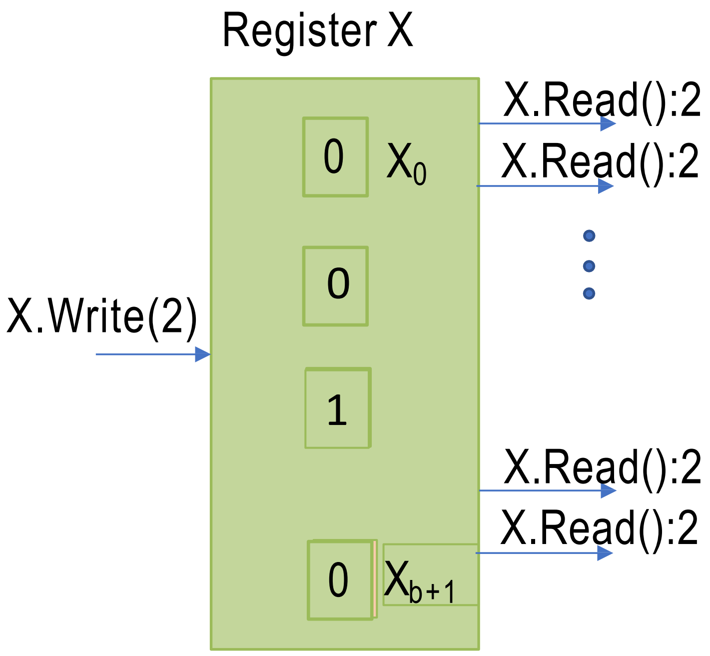
Construction 5:
b-valued SWMR atomic from binary SWMR atomic registers

X: b-valued SWMR atomic register we want to build

Let X_0, …X_b be *b+1* binary SWMR atomic registers initialized with "0" except one with "1"

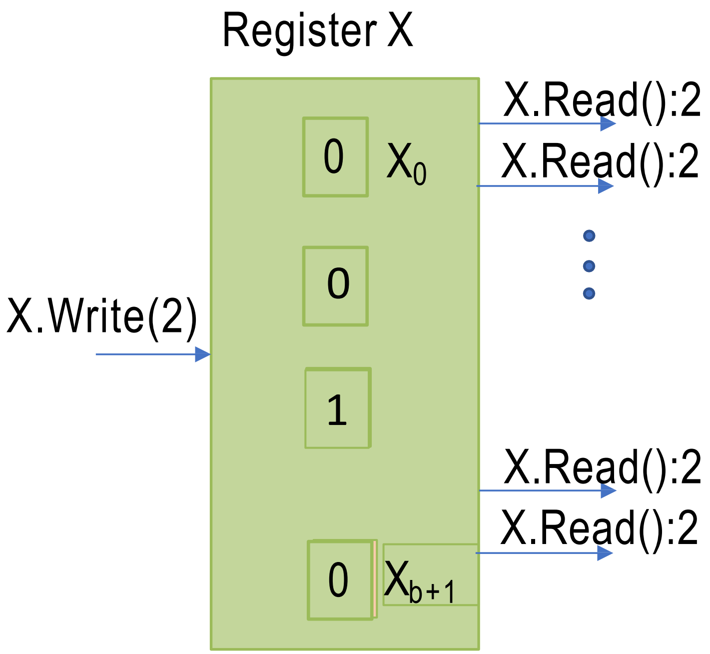
We modify the Read operation of Construction 4 to prevent any old/new inversion phenomena.

When the Read() operation finds a register Xj with "1" then it goes back from j to 0 and returns the smallest register that is set to "1".



Construction 5:
b-valued SWMR atomic from binary SWMR atomic registers

```
when p invokes X.Write(v):
    X_v.write(1)
    for each i in {v-1, .., 0}
        X_i.write(0)

when p_i invokes X.Read():
    up:=0
    while (X_i.Read() ≠ 1) do
        up:=up+1
    j:=up
    for down:=up-1 to 0
        if (Xdown.Read() = 1) then j:=down
    return (j)
```



Construction 5:
b-valued SWMR atomic from binary SWMR atomic registers

Construction 5 implements a b-valued atomic register from a binary atomic one

Proof: For every execution (history) of the algorithm, we define the reading function f as follows:
Let r be a Read() that returned v. Then f(r) is the latest Write operation that updated Xv before the last read() of Xv by r (or the initialization Write() operation w_0 if no such Write() operation exists)
Since r returns v, f(r) writes "1" to Xv
We show that the reading function f satisfies properties A1, A2 and A3

Construction 5:
b-valued SWMR atomic from binary SWMR atomic registers

Construction 5 implements a b-valued atomic register from a binary atomic one

Proof:

- A1 : $\forall$ r: $\neg$(r $\rightarrow_H$ f(r))   (i.e., no read returns a value not written yet)

By definition f(r) is a preceding or concurrent Write() operation. Thus A1 is satisfied

Construction 5:
b-valued SWMR atomic from binary SWMR atomic registers

Construction 5 implements a b-valued atomic register from a binary atomic one

Proof:

- A2 : $\forall$ r in H: ($w \rightarrow_H r$) $\Rightarrow$ ($f(r) = w \vee w \rightarrow_H f(r)$) (i.e., no read returns an overwritten value)

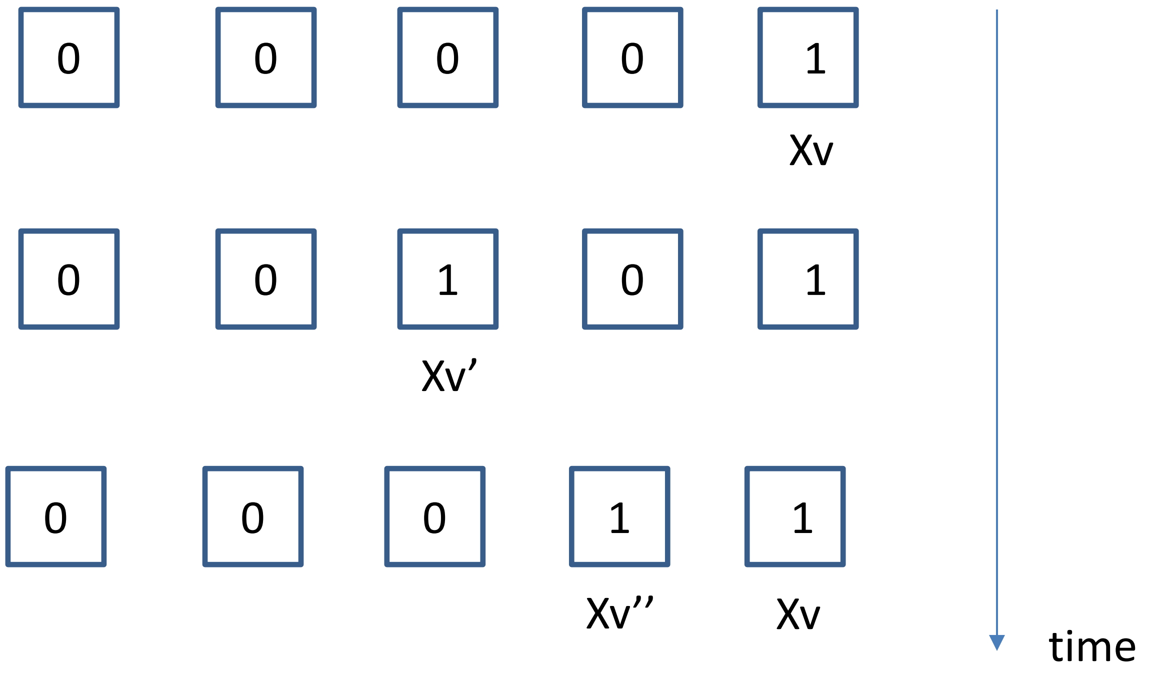
  Suppose by contradiction that it exists some w(v') such that $f(r) \rightarrow_H w(v') \rightarrow_H r/\{v\}$
  By the algorithm, w(v') sets $X_{v'}$ to « 1 » and $X_{v'-1}$, .. X_0 to « 0 ».
  Thus v'<v. Otherwise w(v') would write to Xv between f(r) and r\{v} which contradicts the definition of f(r)
  Since r returns {v} then it must exist a Write w(v'') that sets $X_{v'}$ to « 0 » after that w(v') set it to « 1 » but before r reads it
  By the algorithm, before setting $X_{v'}$ to « 0 », Write(v'') has set $X_{v''}$ to « 1 ».
  By assumption v'' < v. Assuming that w(v'') is the latest such write, before reaching Xv, r should have reached $X_{v''} = 1$. A contradiction

Construction 5:
b-valued SWMR atomic from binary SWMR atomic registers

Construction 5 implements a b-valued atomic register from a binary atomic one

Proof:

- A2 : $\forall$ r in H: $(w \rightarrow_H r) \Rightarrow (f(r) = w \vee w \rightarrow_H f(r))$ (i.e., no read returns an overwritten value)



Construction 5:
b-valued SWMR atomic from binary SWMR atomic registers

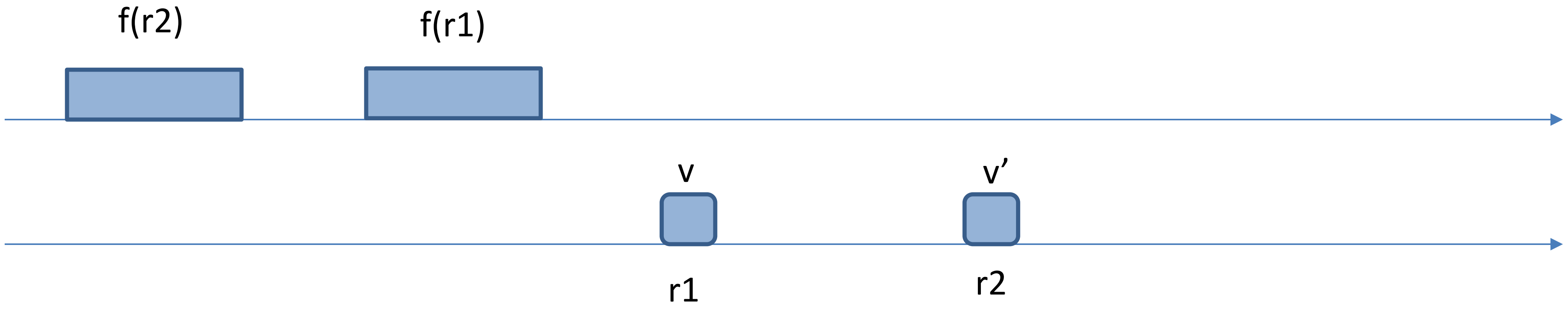
Construction 5 implements a b-valued atomic register from a binary atomic one

Proof:

A3 : $\forall$ r1, r2: ($r1 \rightarrow_H r2$) $\Rightarrow$ ($f(r1) = f(r2) \vee f(r1) \rightarrow_H f(r2)$) (No new/old inversion)

Suppose by contradiction that $f(r2) \rightarrow_H f(r1)$ and $f(r1) \neq f(r2)$. Let r1 / {v} and r2 / {v'}.
Since $f(r1) \neq f(r2)$ we have $v \neq v'$

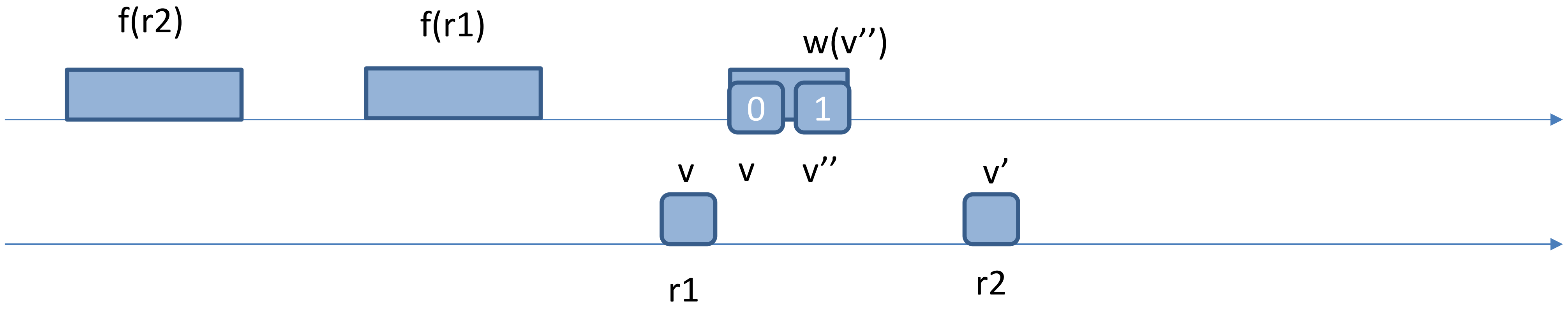
1. v' > v : r2 must have found « 0 » in Xv before returning Xv'



Construction 5:
b-valued SWMR atomic from binary SWMR atomic registers

Suppose by contradiction that f(r2) $\rightarrow_H$ f(r1) and f(r1) ≠ f(r2). Let r1 / {v} and r2 / {v'}.

1. v' > v :
   r2 must have found « 0 » in Xv before returning Xv'=1
   Thus, it exists a write op w(v'') s.t. v < v'' < v' and f(r2) $\rightarrow_H$ w(v'') $\rightarrow_H$ r2, i.e.,
   w(v'') must have set Xv=0 after f(r1) set Xv to 1 but before r2 read Xv=0
   Since w(v'') has set Xv'' = 1 before writing Xv=0, r2 should have returned v''.
   A contradiction.



Construction 5:
b-valued SWMR atomic from binary SWMR atomic registers

Suppose by contradiction that $f(r2) \rightarrow_H f(r1)$ and $f(r1) \neq f(r2)$. Let r1 / {v} and r2 / {v'}.

1. v' < v :

   r1 reads 1 in Xv and then reads 0 in X_{v-1},…, $X_{v'}$, …., X_0 since v > v',

   Since f(r2) has previously set $X_{v'}$ to 1, it must exists another write op that must have set $X_{v'}$ to 0 after f(r2) set $X_{v'}$ to 1 and before r1 reads $X_{v'}$ equal to 0

   Thus when r2 subsequently read 1 in $X_{v'}$, f(r2) is not the last preceding write operation to write in $X_{v'}$

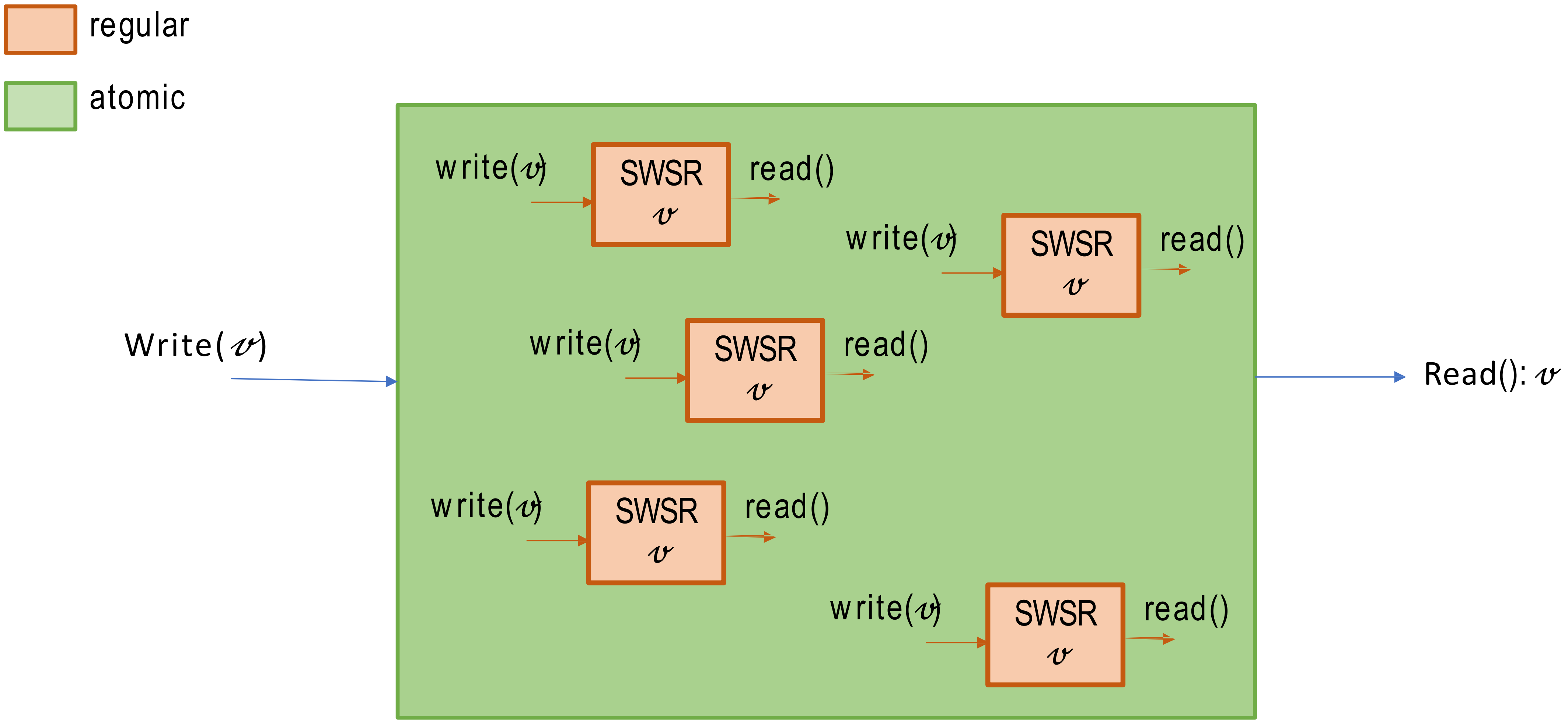
   A contradiction with the definition of theread function f.

Constructions with Unbounded registers

- By using unbounded base registers (i.e., registers of unbounded capacity) we can add sequence numbers: each written value is associated with a sequence number.
- Intuitively it allows us to capture the number of operations that have been performed up to now

- Based on unbounded base registers, we how how to transform
  - A 1W1R regular register into a 1W1R atomic register, then
  - A 1W1R atomic register into a 1WMR atomic register, then
  - A 1WMR atomic register into a MWMR atomic register

Construction 6:
1W1R atomic from 1W1R regular registers



Construction 6:
1W1R atomic from 1W1R regular registers

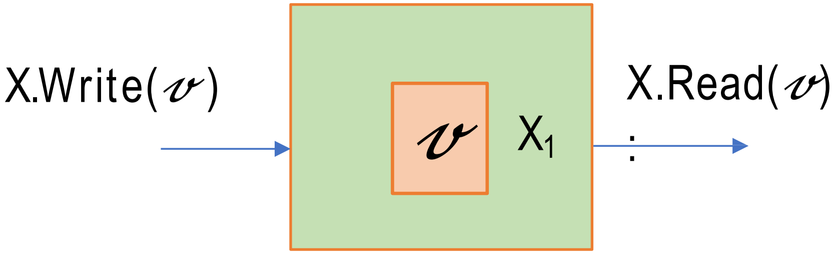
Construction 6
The 1W1R atomic register X uses a 1W1R unbounded regular base register X1

The writer uses 1 local variable *sn* to hold sequence numbers
It is incremented at each new write in X

The reader uses 2 local variables:
- *aux* that spans a read operation. It is made of two fields: a sequence number *aux.sn* and a value *aux.val*
- *last that records the greatest seq number it has ever read in X1 and its associated value*

Construction 6:
1W1R atomic from 1W1R regular registers

```
X.Write(v):  // code executed by p_i
    sn := sn+1
    X_1.write(v,sn)
    return()


X.Read():  // code executed by p_j
    aux := X_1.read()
    if (aux.sn > last_sn) then
        last_sn := aux_sn
        last_val:=aux_val
    return (last_val)    // the value with the highest seq.
                            number returned
```



Construction 6:
1W1R atomic from 1W1R regular registers

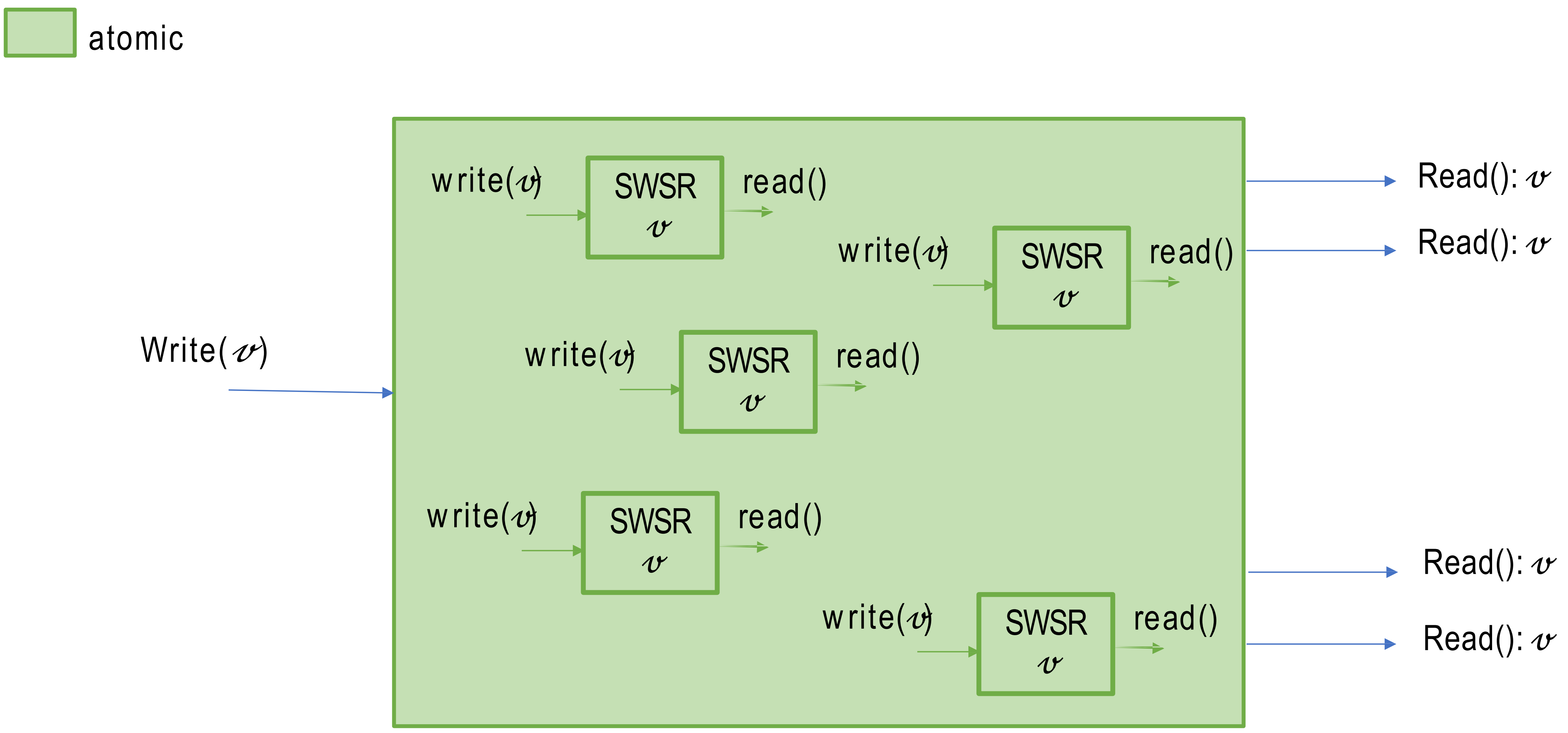
Construction 6 implements a 1W1R atomic register from an unbounded 1W1R regular register

Proof: left as an exercise

Construction 7:
1WMR atomic from 1W1R atomic registers

- We might think that by using multiple SWSR regular registers we might be able to build a SWMR atomic register
- For instance by associating 1 SWSR to each reader and have the writer writes in all of them
- But a fast reader might first see a new concurrently written value while a second reader may read an older value. This is because readers do not know the timestamps of each other and
- time does not grow at the same rate at each reader

Construction 7:
1WMR atomic from 1W1R atomic registers



Construction 7:
1WMR atomic from 1W1R atomic registers

- Idea: All the readers must help each other !

Construction 7:
1WMR atomic from 1W1R atomic registers

Idea: All the readers must help each other !

- Help the others: before returning the read value v, pi informs all the readers that it read v
- Helped by the others: the read value is the one associated with the greatest sequence number ever seen in the base registers

Construction 7:
1WMR atomic from 1W1R atomic registers

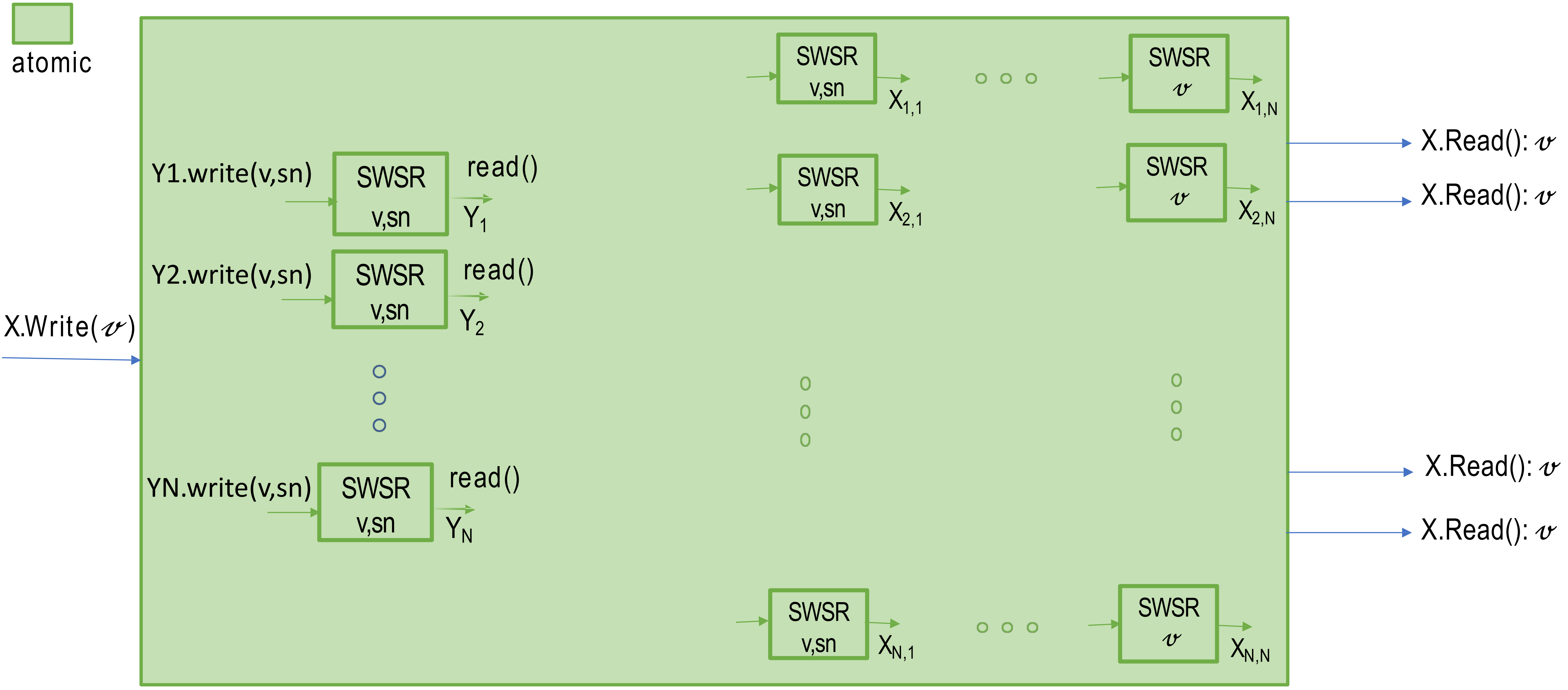
- Requires:

  1. to know the number N of readers

  2. to use N x N 1W1R atomic registers: $X_{k,j}$ (p_k is the reader and p_j is the writer of $X_{k,j}$) to allow all the readers to communicate with each other the new values

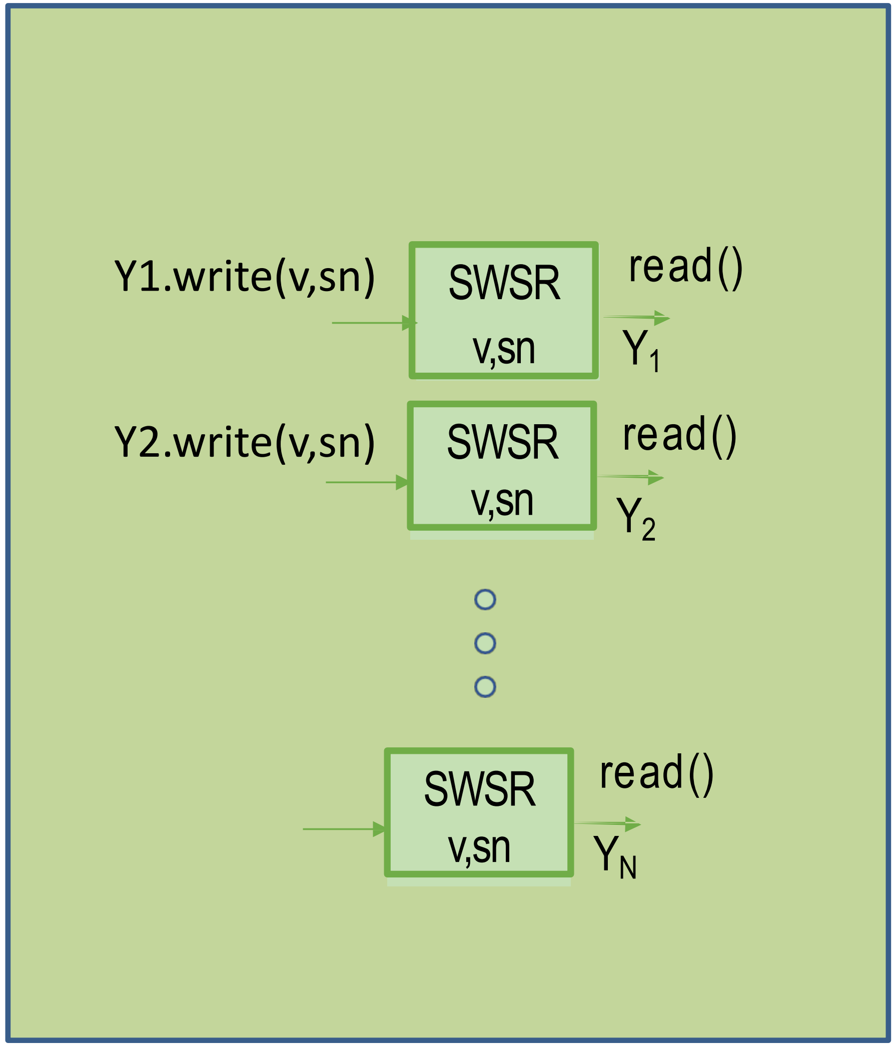
  3. to use N 1W1R atomic registers: Y_j to write the new values

Construction 7:
1WMR atomic from 1W1R atomic registers



Construction 7:
1WMR atomic from 1W1R atomic registers

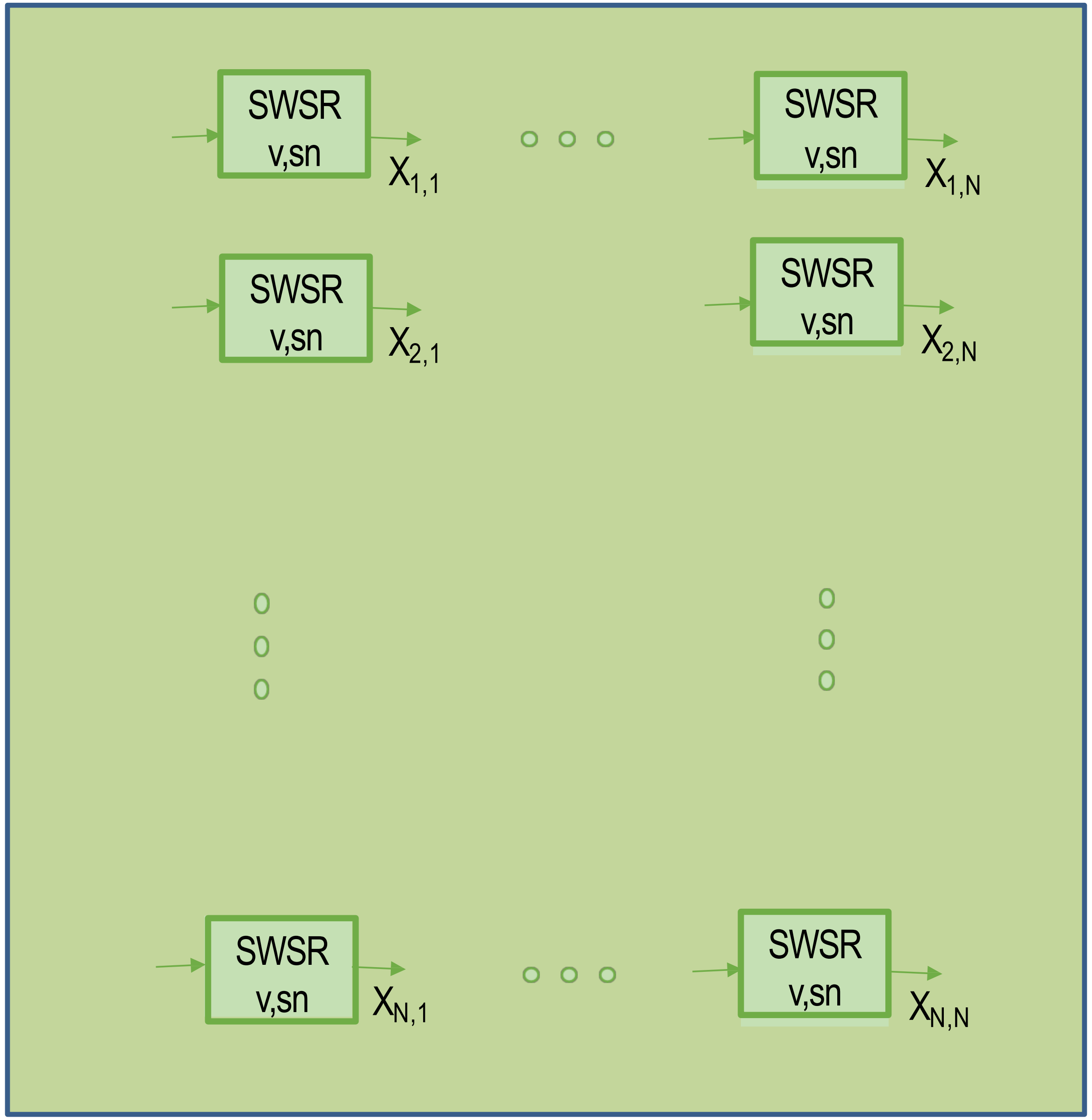
```
X.Write(v): { // code executed by pi

    sn := sn +1
    for j=1 to N  Yj.write(v,sn)
    Return()
}
```



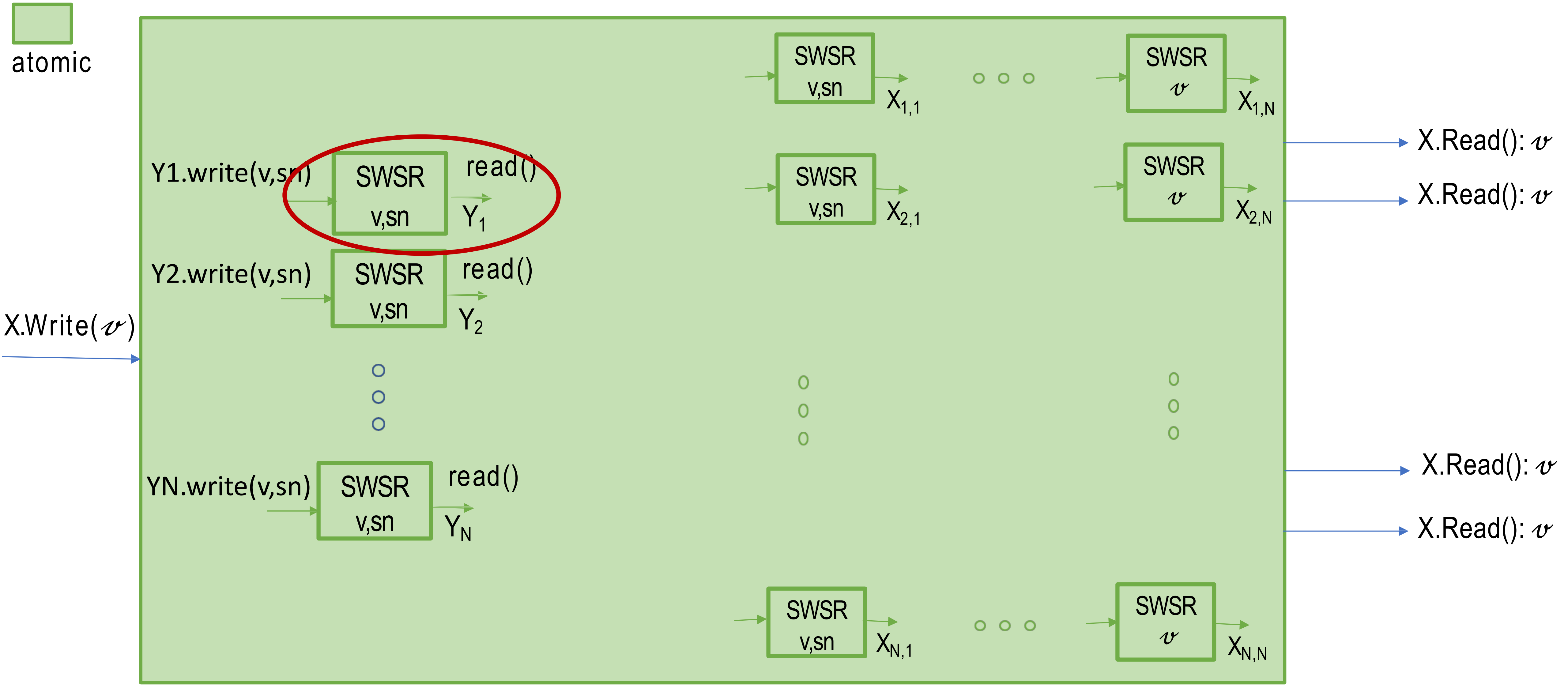
Construction 7:
1WMR atomic from 1W1R atomic registers

X.Read()   // code executed by p_i

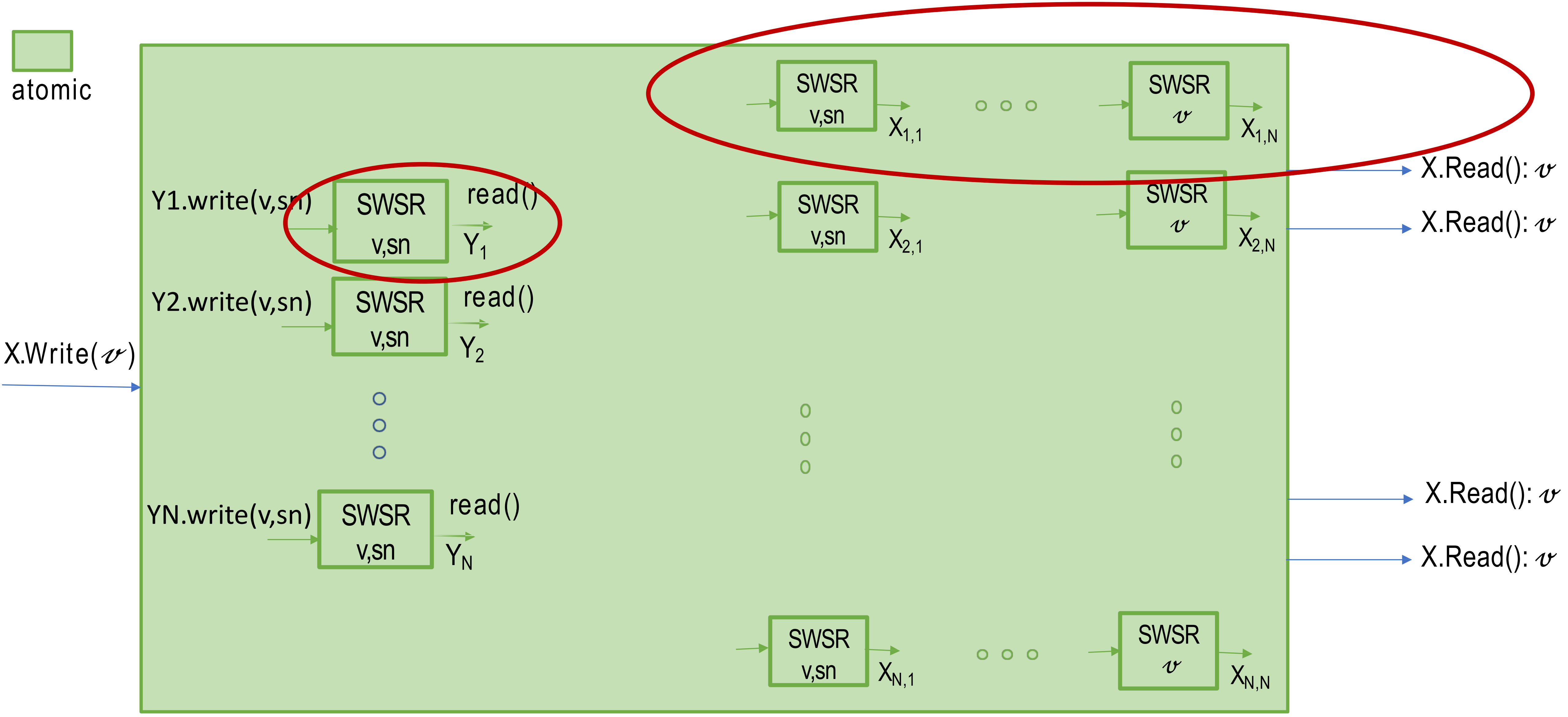
last := Yi.read()
for j=1 to N
    $aux_j := X_{i,j}.read()$
(t_{max}, v) := tuple with largest t
for j=1 to N
    $X_{j,i}.write(t_{max}, v)$
    return (v)



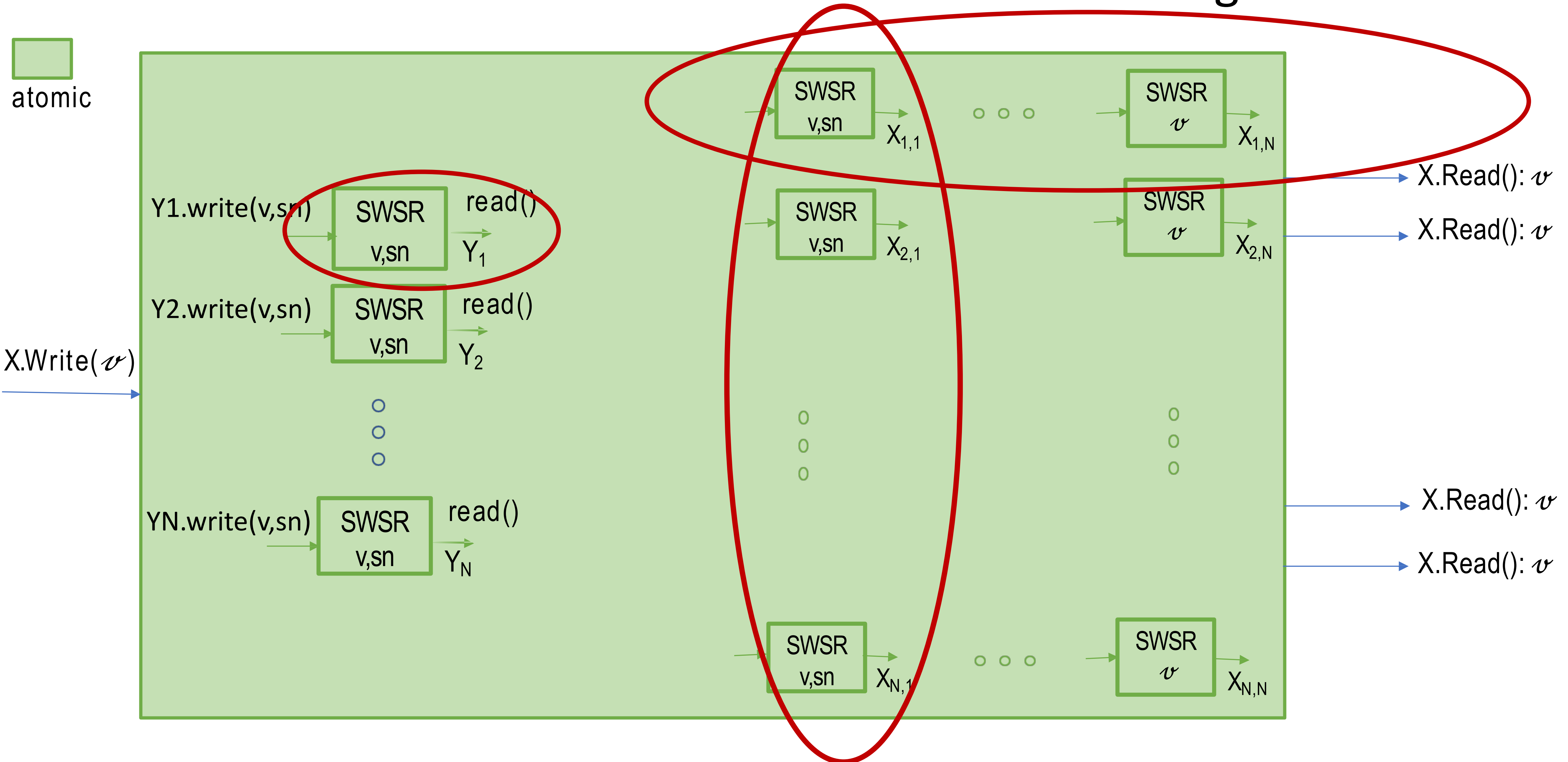
Construction 7:
1WMR atomic from 1W1R atomic registers



Construction 7:
1WMR atomic from 1W1R atomic registers



Construction 7:
1WMR atomic from 1W1R atomic registers



Construction 8:
MWMR atomic from 1WMR atomic registers

- Hint:
  - All writers must determine the “current time”, i.e., the largest timestamp ever used by one of them
  - Write() operation: determine the current time and then apply the write to reader registers

Bibliography

On interprocess communication, Part I and II. Distributed computing. Vol 1, Number 2, pages 77 – 101.