# The Disclosure Power of Shared Objects

Peva Blanchard, Rachid Guerraoui, Julien Stainer, and Igor Zablotchi[⋆]

EPFL

**Abstract.** Shared objects are the means by which processes gather and exchange information about the state of a distributed system. Objects that disclose more information about the system—and thus provide a more centralized view—are therefore more desirable. In this paper, we propose the schedule reconstruction (SR) problem as a new metric for the disclosure power of shared memory objects. In schedule reconstruction, processes take steps which are interleaved to form a schedule; each process needs to be able to reconstruct the schedule up to its last step. We show that objects can be ranked in a hierarchy according to their ability to solve SR. In this hierarchy, stronger objects can implement weaker objects via a SR-based universal construction. We identify a connection between SR and consensus and prove that SR is at least as hard as consensus. Perhaps surprisingly, we show that objects that are powerful in solving consensus—such as compare-and-swap—are not always powerful in their ability to solve SR.

## 1 Introduction

Programming a computing system in a centralized way is significantly more powerful than doing so in a distributed way. The main difficulty of distributed programming comes from the lack of knowledge that a process has about the state of the other processes and the overall state of the system. The more information a process has about the state of the system, the easier it is to write an algorithm for that process to achieve a task in coordination with the other processes. In a distributed system, this information can only be obtained by processes from shared objects. So, intuitively, the more information an object discloses about the rest of the system, the more appealing it is.

In this paper, we propose the schedule reconstruction (SR) problem as a new metric for the disclosure power of shared objects. In order to solve SR, processes in a shared memory system need to be able to accurately identify the interleaving of steps taken by all processes (the schedule). It is easy to see why objects that are able to identify the schedule are desirable. Having knowledge of the schedule basically equates knowing the full system state and thus overcoming the main difficulty of distributed programming, as mentioned above.

We associate a SR number with each object $A$, representing the maximum number of processes of a system in which $A$ can solve SR. The SR number thus allows us to organize shared objects in a hierarchy, with each level corresponding to a SR number.

[⋆] This work has been supported in part by the European ERC Grant 339539 (AOC)

There is a natural connection between disclosure power as measured by the SR number and synchronization power as measured by the consensus number (in the sense of [9]). Intuitively, synchronization is a means of restricting the very large space of executions of a concurrent algorithm, whereas SR is a way of identifying which one of these possible executions actually occurred. At first glance, one would expect that objects with a high reconstruction power (high SR number) should also have a high synchronization power (high consensus number). We confirm this intuition by showing that SR is at least as hard as consensus: the SR number of an object is at most its consensus number.

Due to this connection between SR and consensus, intuition might also predict the inverse relationship to hold true: that objects with a high consensus number should also possess a high SR number. Surprisingly, this is not always the case, as we show in this paper. We prove that compare-and-swap, a very powerful, even universal [9], synchronization primitive, is no more powerful than simple read-write registers in terms of schedule reconstruction.

An object $A$'s position in the SR hierarchy is not only related to its ability to solve the SR problem, but it also determines $A$'s power to implement other shared objects. We show that in the SR hierarchy a stronger object $A$ is always able to implement a weaker object $B$, by providing a universal construction based on SR objects. We also show that $B$ is unable to implement $A$ in such a way that the implementation maintains the same disclosure power as $A$. In other words, implementing a stronger object from a weaker one always entails losing some disclosure power.

To summarize, our contributions are threefold:

- We define the SR problem and the SR number and examine the structure of the SR hierarchy. We prove that there exists at least one class of objects at every level of the hierarchy and at least one object with infinite SR number.
- We identify the relationship between SR and consensus. We show that the former is stronger than the latter and that the consensus number of an object is an upper bound for its SR number.
- We prove the existence of universal objects in the SR hierarchy by providing a SR-based universal construction.

The rest of this paper is organized as follows. In Section 2 we specify the model and define the SR problem and the SR number. In Section 3 we explore the connection between consensus and SR before examining how the SR hierarchy is populated in Section 4. In Section 5, we give a universal construction implementing any shared object from SR objects. We conclude with Section 6.

## 2 Model and Problem Statement

### 2.1 Processes

We consider a set of $n$ processes $P = \{P_1, ..., P_n\}$ that communicate through shared memory using a set of memory access primitives. The processes are executing an algorithm $\mathcal{A}$, which consists of a sequence of shared memory accesses and local steps. We assume local steps to be instantaneous and shared memory steps to be atomic.

An execution of algorithm $\mathcal{A}$ by a set of processes $P$ is modeled by a *schedule* — a finite or infinite sequence of process identifiers which represents the interleaving of steps taken by the processes. A *subschedule* of schedule $S$ is a subsequence of $S$. When describing a schedule, we ignore local steps, so a schedule defines a global total order on the shared memory accesses done by all processes participating in the execution.

We define $S \to t$, where $t \in S$, the $t$-prefix of $S$, to be a subschedule of $S$ consisting of all steps in $S$ up to and including $t$.

### 2.2 Schedule Reconstruction Object

A *schedule reconstruction object* (or *SR object*) provides two methods, `step` and `reconstruct`, neither of which takes any arguments. Basically, a call to `reconstruct` by a process $p$ returns the schedule up to the last `step` call by $p$. The two methods need to satisfy the following conditions:

- the execution of each call to `step` performs exactly one global atomic step and any number of local steps.
- `step` calls may store local values called *observations.* These observations may be used in future `reconstruct` calls.
- `reconstruct` may only be implemented using local steps (including examining the observations gathered by the calling process) and shared memory accesses *that do not modify the state of shared memory* (such as reads).
- a call to `reconstruct` by process $p$ returns the schedule as a mapping from step numbers to process ids (more precisely, a mapping from $\{1, 2, ..., s\}$ to $P$, where $s$ is the global number of `step` calls up to and including the last call to `step` by $p$) or an empty mapping if there are no `step` calls by $p$ preceding the `reconstruct` call.

We are interested in wait-free implementations of SR objects that correctly reconstruct *any possible schedule* (any interleaving of `step` calls). We call any such implementation a *SR algorithm.*

A class $C$ of objects solves the $n$-process schedule reconstruction problem if there exists a SR algorithm $\mathcal{A}$ that solves $n$-process schedule reconstruction using any number of objects of class $C$ and any number of atomic registers.

We define the *schedule reconstruction number* (or *SR number*) of a class $C$ to be the largest $n$ for which that class solves $n$-process schedule reconstruction. If no largest $n$ exists, we say that the SR number of the class is *infinite.*

## 3 SR and Consensus

In this section, we first give a short background on the consensus problem, before establishing the relationship between consensus and SR.

### 3.1 Background

Consensus is a fundamental problem in distributed systems [10]. In its simplest form, consensus requires a set processes to agree on a value. Despite its simplicity, this is a difficult problem: it has been shown impossible to solve in asynchronous message-passing systems [8] and in asynchronous shared memory systems with only read-write memory [9,13]. In what follows, we consider the problem in the context of shared memory.

We define the consensus problem more precisely [10]: an algorithm solves consensus if it implements a linearizable [11] wait-free *consensus object*. A consensus object has a single method *propose* that accepts a *proposed value* and returns a *decided value*. Each process calls the *propose* method of a consensus object at most once. A correct implementation of a consensus object needs to satisfy the following properties (besides wait-free termination and linearizability):

**Validity** if a process decides a value $v$, then $v$ was proposed by some process
**Agreement** if a process decides a value $v$, then no process decides a value $v' \neq v$

In a system of $n$ processes, we say that a class of objects $C$ implements consensus if there exists an implementation of a consensus object from (any number of) objects of class $C$ and atomic registers. The *consensus number* of a class $C$ is the largest number of processes $k$ for which $C$ implements consensus (if no largest $k$ exists, the consensus number is infinite). The consensus numbers (CN) of well-known classes have been studied: read-write registers and snapshot [1,3,5,7] objects have CN 1; queues, stacks, fetch-and-increment, get-and-set and more generally, read-modify-write objects in *Common2* [4] have CN 2; $m$-assignment registers have CN $m$; compare-and-swap (among others) has infinite consensus number.

Another significant notion related to consensus is that of a *universal construction* [2,6,9,14]. It has been shown that any object with a sequential specification can be implemented from consensus objects and registers (such an implementation is called a universal construction).

Together, the concepts of consensus number and universal construction define a consensus hierarchy, with one level corresponding to each consensus number. The consensus hierarchy provides a definitive answer to the crucial question "Does A implement B in a system of $n$ processes?", for any two classses A and B with known consensus numbers. This answer comes from two major results concerning the hierarchy: a positive one and a negative one. The positive result stems from the universal construction and states that in a system of $n$ processes, any object with CN at least $n$ can implement any other object. The negative result is a direct consequence of the definition of consensus number and states that in a system of $n$ processes, it is impossible for an object with CN lower than $n$ to implement an object with CN at least $n$.

### 3.2 SR Is at Least as Hard as Consensus

Much like the consensus number, the schedule reconstruction number establishes a hierarchy on synchronization primitives. In what follows, we explore the structure of the SR hierarchy and highlight similarities with the consensus hierarchy—it is infinite and populated at every level—as well as differences therewith—the compare-and-swap primitive is at the bottom of the SR hierarchy, whereas it is at the top of the consensus hierarchy.

A first interesting observation is that SR is at least as hard as consensus:

**Theorem 1.** *Any class $C$ of objects that solves $n$-process SR also solves $n$-process consensus.*

*Proof.* Let $\mathcal{A}$ be an algorithm solving $n$-process SR using only objects of class $C$ and atomic registers. Now, consider the consensus protocol in Algorithm 1. Each process writes its proposed value in a single-writer, multi-reader register. Then, each of the $n$ processes calls `step` once and then calls `reconstruct`. Thus, every process knows the schedule and is able to decide on the value proposed by the process which was scheduled first.

```
Shared:
    sr: a SR object
    announce[n]: an array of MRSW registers

propose(value):
    announce[myId] = value
    sr.step()
    schedule = sr.reconstruct()
    leader = schedule.get(1)
    decidedValue = announce[leader]
    return decidedValue
```

Algorithm 1: Consensus from SR

As an immediate consequence of the above result, we have the following.

**Corollary 1.** *The SR number of a class $C$ is at most equal to its consensus number.*

## 4 The SR Hierarchy

We now examine specific classes of objects with respect to their ability to solve the SR problem.

### 4.1 Atomic Registers

From Corollary 1, we know that the SR number of atomic registers is at most 1 (since they have consensus number 1). Since the schedule of a 1-process execution is always equal to the sequence of shared memory steps of the algorithm being executed, atomic registers trivially solve 1-process SR, meaning that they have SR number 1.

## 4.2 Fetch-and-increment

Fetch-and-increment objects store an integer value and provide a single method, `getAndIncrement`, which increments the value and returns the old value.

Fetch-and-increment objects have consensus number 2. Thus, using Corollary 1, this class has SR number at most 2. We will now show that they have SR number exactly 2.

**Theorem 2.** *Fetch-and-increment has SR number 2.*

*Proof.* Consider the pseudocode for 2-process SR shown in Algorithm 2. The two processes share a fetch-and-increment object which initially has value 0. Each process simply executes a loop calling `getAndIncrement` on the shared object and appending the result to a local list of observations.

```
Shared:
    fai: a fetch-and-increment object,
        initially 0

Local:
    obs: a list of integers

step():
    val = fai.getAndIncrement()
    obs.append(val)

reconstruct():
// returns the schedule as a mapping from
    step numbers to process ids
    schedule = {}

    for step from 1 to obs.size():
        schedule.put(obs.get(step), myId)

    stepsCount = obs.lastItem()
    for step from 1 to stepsCount:
        if !schedule.hasKey(step):
            schedule.put(step, otherId)

    return schedule
```

Algorithm 2: SR using Fetch-and-increment

This algorithm solves 2-process schedule reconstruction. To see this, note that the fetch-and-increment object will return consecutive values to the two processes in the order that they are scheduled. As such, each process will have a list of observations consisting of strictly increasing values. Any gaps in this list (non-consecutive values) correspond to steps taken by the other process.

More precisely, consider the observation list of process $p_i$, $i \in \{1, 2\}$ immediately after performing its $t^{\text{th}}$ step. It is of the form $O_i = \{v_1^i, v_2^i, ..., v_t^i\}$, with

$v_1^i < v_2^i < ... < v_t^i$. Then $p_i$ is able to reconstruct the schedule up to its $t^{\text{th}}$ step, as follows: steps $O_i$ of the schedule were performed by $p_i$ (itself) and steps $\{0, 1, ...v_t^i\} \setminus O_i$ were performed by $p_{1-i}$ (the other process).

## 4.3 Compare-and-Swap: a Surprising Result

In this section, we show that the SR number of compare-and-swap (CAS) is (exactly) 1. We know that it is (trivially) at least 1, by the same argument used for atomic registers. It remains to show that it is also at most 1. We do this through several intermediate steps.

**Definition 1.** *An invisible step $t$ is a shared memory access performed by a process $p$ during a call to* `step` *such that for any process $q \neq p$, the shared (thus observable) system state is identical before and after $p$ executes $t$.*

**Lemma 1.** *Let $\mathcal{A}$ be a SR algorithm for some number $n > 1$ of processes. Then, no execution of $\mathcal{A}$ contains an invisible step.*

*Proof.* Assume by contradiction that there exists an execution of $\mathcal{A}$ which contains for some process $p$ an invisible step $t$, and let $S$ be the schedule of this execution. Let $S'$ be the $t$-prefix of $S$ and $S''$ be $S' \setminus t$. $S'$ and $S''$ are both legal schedules for $\mathcal{A}$ (because they were obtained by removing steps from $S$) and since $t$ is an invisible step, $S'$ and $S''$ are indistinguishable for any process $q \neq p$. As such, a `reconstruct` call by $q$ after $t$ will not be able to decide whether to include $t$ in the returned schedule, contradicting the assumption that $\mathcal{A}$ is a SR algorithm.

*Remark 1.* Atomic register reads are invisible steps.

This is because atomic register reads do not modify the observable state of the system. For the same reason, the following is also true.

*Remark 2.* Failed CAS operations are invisible steps.

Therefore, an algorithm solving the SR problem may not contain simple reads, nor may any of its CAS calls fail.

**Theorem 3.** *CAS has SR number at most 1.*

*Proof.* Towards a contradiction, assume that there exists an algorithm $\mathcal{A}$ for 2-process SR using only CAS objects and registers. By the third observation above, we know that $\mathcal{A}$ cannot contain any reads. Consider the first step of $\mathcal{A}$ for each process. There are four possibilities, each of which can be denoted by a tuple $(t_1, t_2)$, where $t_1$ is the first step of $p_1$ and $t_2$ is the first step of $p_2$, and $t_1, t_2 \in \{W, CAS\}$, where $W$ represents an atomic write and $CAS$ represents a compare-and-swap operation. We now consider each of the four possibilities.

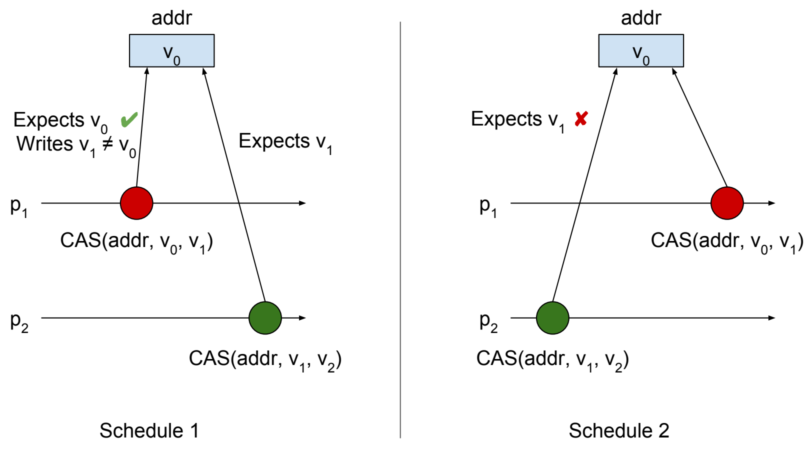

Fig. 1: Two consecutive CAS operations on the same address. If both operations succeed in Schedule 1, then the CAS by $p_2$ will fail in Schedule 2, becoming an invisible step.

1. $(W, W)$. If the two writes occur to different memory locations, then it is impossible for the processes to determine the order in which they occurred (they commute). So the two writes must occur to the same memory location. Consider two schedules: (1) $p_1$ writes first, then $p_2$ writes and calls `reconstruct` and (2) $p_2$ writes and calls `reconstruct` before $p_1$ takes a step. The `reconstruct` calls will not be able to distinguish between the two schedules, because they have no way of knowing whether $p_1$ has already perfored its write or not.
2. $(W, CAS)$. Immediately after $p_1$ executes its write, it has no information whether $p_2$ performed its CAS yet or not, thus it cannot reconstruct the schedule.
3. $(CAS, W)$. This case is symmetric with the $(W, CAS)$ case, so now $p_2$ is unable to reconstruct the schedule immediately after its first write.
4. $(CAS, CAS)$. Note that both CAS steps have to be successful, otherwise they would be invisible steps. Also note that both CAS steps need to be performed on the same memory location (call it *addr*), otherwise it would be impossible to determine the order in which they occur. Denote by $v_0$ the value of *addr* before any process takes a step.
   Consider the schedule in which $p_1$ takes its first step before $p_2$. The CAS by $p_1$ is successful, so it expects $v_0$. Call $v_1$ the value written by the CAS of $p_1$. We must have $v_1 \neq v_0$, otherwise the CAS would be an invisible step. Now the CAS by $p_2$ must also be successful, so it expects to find $v_1$.
   Now consider the schedule in which $p_2$ takes the first step, followed by $p_1$. The CAS by $p_2$ expects $v_1 \neq v_0$, but finds $v_0$, so it fails, thus becoming an invisible step, a contradiction.

## 4.4 Multiple Atomic Append: Every Level is Populated

An *append register* is similar to a regular register, except that every write appends its value to the current value of the register, instead of overwriting it. More precisely, an append register stores a value $v$ and provides two methods: `read`, which returns the current value $v$, and `append`, which takes as argument a value $v_{arg}$ and appends it to the current value $v$ (thus updating the current value to $v \cdot v_{arg}$). A $k$-writer append register is an append register from which any number of processes can read but to which only $k$ processes can append. Interestingly, append registers have been studied in a Byzantine setting as well [12].

**Theorem 4.** *$k$-writer append registers have SR number $s = k$.*

*Proof.* First, we show that $s \geq k$. We provide a SR algorithm for $k$ processes using a shared $k$-writer append register $r$ (Algorithm 3). At step $j$, a process simply appends its process id to $r$. A reconstruct call by process $p_i$ after its $j^{\text{th}}$ step starts by reading $r$ and discarding any values after its last appended value (since reconstruction only needs to be up to the $j^{\text{th}}$ step of $p_i$). What remains is an encoding of the order in which the processes appended their ids to $r$, and thus an encoding of the schedule.

```
Shared:
    r: k-writer append register
Local:
    myStep: integer, initially 0

step():
    myStep += 1
    itemToAppend = (myID, myStep)
    r.append(itemToAppend)

reconstruct()
    contents = r.read()
    schedule = {}
    for each item in contents:
        (id, step) = item
        schedule.put(step, id)
        if id == myId and step == myStep
            break
    return schedule
```

Algorithm 3: $k$-SR from $k$-append registers

It only remains to show that $s < k + 1$. Assume there exists a SR algorithm for $k + 1$ processes using only $k$-writer append registers and atomic read-write registers. We consider the first step of the algorithm for each process. Similarly to the proof of Theorem 3, if some process $p$ reads or writes to a read-write register as its first `step` call, $p$ will be unable to reconstruct the schedule right after this first step. So all processes must access append registers during their first step.

Then, because there are $k+1$ processes but the append registers only support $k$ writers, there must exist two processes $p_i$ and $p_j$ which do not write to the same append register for their first step. Therefore, $p_i$ and $p_j$ cannot distinguish between executions in which the first step of $p_i$ takes effect before the first step of $p_j$ and executions in which the first step of $p_j$ takes effect before the first step of $p_i$, a contradiction.

### 4.5 `SWAP3`: the Hierarchy is Infinite

We define a new primitive called `SWAP3`. `SWAP3` takes three arguments `a`, `b` and `c`. It atomically replaces the value of `c` with the value of `b` and the value of `b` with the value of `a`.

```
SWAP3(a, b, c):
    c = b
    b = a
```

Algorithm 4: `SWAP3` sequential specification

**Theorem 5.** `SWAP3` *has SR number* $\infty$.

*Proof.* Intuitively, an algorithm that solves SR for any number of processes using `SWAP3` works as follows (Algorithm 5). The processes maintain a shared linked list which encodes the schedule. Each node in the list represents one step by one process. The processes share a pointer `head` pointing to first node in the list. As is usual with linked lists, each node has a `next` field pointing to the next node in the list, and the last node points to a special value $\perp$. The list encodes the schedule in reverse order: `head` points to the most recent step and each step points to the step that immediately precedes it in the schedule.

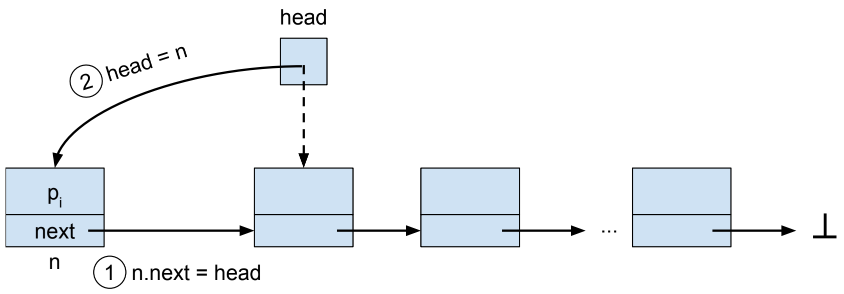

Fig. 2: Appending a new node to a list with a single `SWAP3` call.

A `step` call by a process $p_i$ prepares a new node with its process id and the current step number (local steps) and appends it to the head of the list (a single global step using `SWAP3`). As shown in Figure 2, appending a node `n` to the list using `SWAP3` works by atomically updating the `next` field of `n` to point to `head` and `head` to point to `n`: `SWAP3(n, head, n.next)`. In order to reconstruct the schedule up to its last global step, a process $p_i$ starts from the head of the list and traverses the list using `next` pointers until it reaches the $\perp$ value: the schedule is given by the list nodes in reverse order, excluding any steps that may have been appended after the last step of $p_i$.

```
Structure used:
    node = (id, next)
Shared:
    head: pointer to node

step():
    n = new node
    n.id = myId
    SWAP3(n, head, n.next)

reconstruct():
    schedule = {}

    // traverse list up to my last step
    curNode = head
    while curNode != ⊥ && curNode.id != myId:
        curNode = curNode.next

    // return empty schedule if no steps so far
    if curNode == ⊥:
        return schedule

    // count steps from my last step to beginning
       of execution
    firstNode = curNode
    stepCount = 0
    while curNode != ⊥:
        stepCount += 1
        curNode = curNode.next

    // add steps to schedule
    curNode = firstNode
    while curNode != ⊥:
        schedule.put(stepCount, curNode.id)
        stepCount -= 1
        curNode = curNode.next

    return schedule
```

Algorithm 5: SR from `SWAP3`

A different way of seeing that that `SWAP3` has SR number $\infty$ is by noting that the list encoding the schedule as described above is essentially an $\infty$-writer append register.

## 5 A SR-based Universal Construction

In the previous sections, we have shown that shared objects can be ranked in a hierarchy according to their ability to solve the SR problem (Table 1). In this section, we examine the relationships between the levels in this hierarchy, namely the power of objects at higher levels to implement objects at lower levels.

| **SR number** | **Object** |
|---|---|
| 1 | atomic registers, compare-and-swap |
| 2 | fetch-and-increment |
| $\vdots$ | $\vdots$ |
| $k$ | $k$-writer append register |
| $\vdots$ | $\vdots$ |
| $\infty$ | SWAP3 |

Table 1: The SR hierarchy

We give two main results: a positive one and a negative one. The positive result states that stronger objects (in the SR sense) can implement weaker objects, whereas the negative result states that weaker objects cannot implement stronger objects in a way that preserves reconstructibility.

**Definition 2.** *An object $A$ implements an object $B$ if there exists a wait-free linearizable implementation of $B$ that only uses (any number of) objects of type $A$ and atomic registers.*

**Definition 3.** *An object $A$ $k$-implements an object $B$ if (1) $A$ implements $B$ and (2) the implementation performs at most $k$ shared memory accesses per call to $B$'s methods.*

We begin with the positive result: in a system of $n$ processes, given any object $A$ with SR number $\geq n$ and any deterministic object $B$, $A$ implements $B$.

By definition of SR number, $A$ can be used to (1-)implement SR objects in a system of $n$ processes. Thus, it suffices to show that $SR$ objects implement $B$.

The construction used to implement $B$ from an SR object and atomic registers is shown in Algorithm 6. Intuitively, processes use the SR object to determine the order in which their invocations take effect and then use this information to simulate the execution on local copies of $B$.

The construction works as follows. Each process $p$ keeps a private copy of $B$. In order to invoke a method on $B$, $p$ first announces the method and associated arguments (if any) in a shared announce array (lines 13–14). Then, $p$ takes a step on the SR object and reconstructs the schedule (lines 15–16). Next, $p$ applies on its private copy of $B$ all pending invocations (including its own), in the order specified by the reconstructed schedule (lines 17–26). Finally, $p$ returns, with the return value of the last simulated invocation, since the last step of a reconstructed schedule is always the calling process's latest step.

```
1  Shared:
2      sr: a n-SR object
3      announce[1..n][1..∞]: n infinite arrays of registers
           where processes announce their invocation arguments
4
5  Local:
6      localB: a copy of B
7      myInvocs: the number of invocations announced by this
           process
8      perf: the number of invocations simulated by this process
9      curStep[1..n]: an array of registers, containing the next
            step this process need to simulate from each of the
           other processes.
10
11 invoc(method, args):
12     myInvocs += 1
13     announce[myId][myInvocs] = (method, args)
14     sr.step()
15     sched = sr.reconstruct()
16     for step from perf to sched.numSteps():
17         curProc = sched.get(step)
18         (curMethod, curArgs) = announce[curProc][curStep[
               curProc]]
19         curStep[curProc]++
20
21         // perform step on localB
22         retVal = localB.invoc(curMethod, curArgs)
23
24         // update perf
25         perf += 1
26
27     return retVal
```

Algorithm 6: SR universal construction

We have just shown that in the SR hierarchy, as in the consensus hierarchy, there exist objects that are *universal.* Given sufficiently many of them, any object with a sequential specification can be implemented in a wait-free linearizable way.

We now turn to the negative result: in a system of $n$ processes, given any object $A$ with SR number $\geq n$ and any object $B$ with SR number $< n$, $B$ cannot 1-implement $A$.

Towards a contradiction, assume that $B$ can 1-implement $A$ in a system of $n$ processes. Since $A$ has SR number $\geq n$, there exists an implementation of a SR object from $A$ and atomic registers. By replacing $A$ in this implementation with its 1-implementation from $B$, we will have obtained a valid implementation of an SR object from $B$, a contradiction of the fact that $B$'s SR number is less than $n$.

Note that this negative result does not contradict the universality of objects in the sense of consensus, which states that an object with consensus number at least $n$ can implement any object in a system of $n$ or less processes. Our negative result states that objects with lower SR number cannot *1-implement* objects with higher SR number. So, for instance, compare-and-swap has infinite consensus number, so it can implement any object, but it has SR number 1, so it cannot *implement in a single step* any object with SR number larger than 1 (e.g., fetch-and-increment) in a system of 2 or more processes. In other words, no such object can be implemented from compare-and-swap in such a way that the implementation has the same SR number as the abstract object.

## 6 Conclusion

In this paper, we propose the schedule reconstruction problem and the SR number as a new measure for the disclosure power of objects in shared memory systems. Objects can be organized in a dense hierarchy where strong objects implement weaker objects via a universal construction based on SR. Furthermore, we identify a link between SR and consensus and show that SR is at least as hard as consensus. Finally, we evaluate the SR number of well known objects and show that universal consensus objects are not always universal SR objects.

In terms of future work, this paper opens two interesting directions, in the form of alternative measures for the schedule disclosure ability of shared objects. First, instead of the current formulation of SR, which ranks objects discretely (an object either can or cannot solve the SR problem for a given number of processes), a different SR formulation could rank objects on a continuous scale: given a number of processes and any number of objects of a certain type, how much information (in bits) can be recovered about the schedule? Second, we could also introduce a notion of reconstruction latency. Instead of requiring `reconstruct` calls to fully recreate the schedule up to the last preceding `step` call, we could allow a certain number of preceding steps to not be reconstructed. In other words, for a given number of processes, some object A might not be able to fully solve SR but still be able to reconstruct the schedule up to some older `step` call.

## References

1. Afek, Y., Attiya, H., Dolev, D., Gafni, E., Merritt, M., Shavit, N.: Atomic snapshots of shared memory. Journal of the ACM (JACM) 40(4) (1993)
2. Afek, Y., Dauber, D., Touitou, D.: Wait-free made fast. STOC 1995
3. Afek, Y., Stupp, G., Touitou, D.: Long-lived and adaptive atomic snapshot and immediate snapshot. PODC 2000
4. Afek, Y., Weisberger, E., Weisman, H.: A completeness theorem for a class of synchronization objects. PODC 1993
5. Anderson, J.H.: Composite registers. Distributed Computing 6(3) (1993)
6. Anderson, J.H., Moir, M.: Universal constructions for multi-object operations. PODC 1995
7. Borowsky, E., Gafni, E.: Immediate atomic snapshots and fast renaming. PODC 1993
8. Fischer, M.J., Lynch, N.A., Paterson, M.S.: Impossibility of distributed consensus with one faulty process. Journal of the ACM (JACM) 32(2) (1985)
9. Herlihy, M.: Wait-free synchronization. ACM Transactions Programming Languages and Systems 13(1) (1991)
10. Herlihy, M., Shavit, N.: The Art of Multiprocessor Programming, Revised Reprint. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1st edn. (2012)
11. Herlihy, M.P., Wing, J.M.: Linearizability: A correctness condition for concurrent objects. ACM Transactions on Programming Languages and Systems 12(3) (1990)
12. Imbs, D., Rajsbaum, S., Raynal, M., Stainer, J.: Read/write shared memory abstraction on top of asynchronous byzantine message-passing systems. Journal of Parallel and Distributed Computing 93 (2016)
13. Loui, M., Abu-Amara, H.: Memory requirements for agreement among unreliable asynchronous processors. Advances in Computing Research 4 (1987)
14. Moir, M.: Laziness pays! Using lazy synchronization mechanisms to improve non-blocking constructions. PODC 2000