Hyperproperty-Preserving Register Specifications

Yoav Ben Shimon
Tel Aviv University, Israel

Ori Lahav
Tel Aviv University, Israel

Sharon Shoham
Tel Aviv University, Israel

Abstract

Reasoning about *hyperproperties* of concurrent implementations, such as the guarantees these implementations provide to randomized client programs, has been a long-standing challenge. Standard linearizability enables the use of *atomic specifications* for reasoning about standard properties, but not about hyperproperties. A stronger correctness criterion, called *strong linearizability*, enables such reasoning, but is rarely achievable, leaving various useful implementations with no means for reasoning about their hyperproperties. In this paper, we focus on registers and devise *non-atomic specifications* that capture a wide-range of well-studied register implementations and enable reasoning about their hyperproperties. First, we consider the class of *write strong-linearizable* implementations, a recently proposed useful weakening of strong linearizability, which allows more implementations, such as the well-studied single-writer `ABD` distributed implementation. We introduce a simple shared-memory register specification that can be used for reasoning about hyperproperties of programs that use write strongly-linearizable implementations. Second, we introduce a new linearizability class, which we call *decisive linearizability*, that is weaker than write strong-linearizability and includes multi-writer `ABD`, and develop a second shared-memory register specification for reasoning about hyperproperties of programs that use register implementations of this class. These results shed light on the hyperproperties guaranteed when simulating shared memory in a crash-resilient message-passing system.

2012 ACM Subject Classification Theory of computation → Concurrency; Theory of computation → Program specifications; Theory of computation → Distributed algorithms; Theory of computation → Concurrent algorithms

Keywords and phrases Hyperproperties, Concurrent objects, Distributed objects, Linearizability, Strong linearizability, Simulation

Digital Object Identifier 10.4230/LIPIcs.DISC.2024.1

Related Version An extended version of the paper is available at [21].

Funding This work is supported by the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation programme (grant agreement no. [759102-SVIS] and 851811) and the Israel Science Foundation (grant number 2117/23 and 814/22).

1 Introduction

Linearizability [17] is a widely accepted correctness criterion for concurrent and distributed implementations of objects, allowing clients of an object to pretend that they use an atomic abstraction thereof, whose behaviors are much easier to understand [13]. The observational refinement between a linearizable implementation and its atomic specification is, however, restricted to reasoning about reachability of 'bad' states. Dealing with more intricate properties, such as the ability of an adversary to control the probability distribution of the results of an object's methods, reveals that a linearizable implementation may manifest behaviors exceeding those permissible by the atomic specification [14]. In the terminology of [3], linearizability ensures preservation of safety properties but fails to maintain *hyperproperties*, which are properties of *sets* of executions, rather than individual executions. These properties allow one to express security guarantees, such as noninterference, as well as probability distributions on program outcomes [9].

The preservation of hyperproperties of concurrent implementations, a.k.a. *strong* observational refinement, necessitates a more strict connection between the implementation and its atomic specification, known as *strong linearizability* [14], which is equivalent to (a certain form of) forward simulation between the implementation and the atomic specification [3, 12]. Many implementations are, however, known to be *non*-strongly linearizable, leaving us with no means to reason about hyperproperties of programs that use these implementations by assuming simpler abstractions. In particular, the well-studied `ABD` implementation, which shows how shared memory can be simulated in a crash-tolerant message-passing system [2], is not strongly linearizable, and, in fact, a strongly linearizable implementation with similar guarantees does not exist [5].

Focusing on registers and observing that strong linearizability is hardly achievable, Hadzilacos et al. [15] recently proposed a weakening of strong linearizability, called *write strong-linearizability*, which captures more implementations. This includes single-writer `ABD`, and, in fact, as shown in [15], every linearizable implementation of a *single-writer* register. We are left, however, with substantial gaps: how should one reason about hyperproperties of programs that use write strongly-linearizable register implementations? and what can be said about existing non-strongly-linearizable implementations of *multi-writer* registers?

The current work aims to address these gaps. Inspired by Attiya and Enea [3], who propose to reason about hyperproperties of programs that use non-strongly-linearizable implementations by using simpler (albeit non-atomic) implementations related to them by strong observational refinement, we present a simple (but necessarily not atomic) specification of a shared multi-writer register, which we call `WSR` (for "Write Strong Register") that can be used for reasoning about hyperproperties of programs that use any write strongly-linearizable implementation (including single-writer `ABD`). To do so, we prove that every write strongly-linearizable implementation has a forward simulation to `WSR`, and utilize the correspondence between forward simulation and strong observational refinement (preservation of hyperproperties). Moreover, since write strong-linearizability is downward closed w.r.t. forward simulation and `WSR` is write strongly-linearizable, one can also prove write strong-linearizability for a given implementation by establishing a forward simulation to `WSR`, which may be more amenable to automatic/machine-assisted proofs than a direct proof. Drawing an analogy to complexity theory, we refer to `WSR` as a *complete* implementation for the class of write strongly-linearizable register implementations: `WSR` is write strongly-linearizable and every write strongly-linearizable implementation has a forward simulation to `WSR`.

As for multi-writer registers, we present a second specification of a shared register that is 'complete' for a family of implementations that admit a weakening of write strong-linearizability, which we call *decisive linearizability.* In particular, we show that multi-writer `ABD` [19] belongs to this family. Thus, the complete implementation, which we call `DR` (for "Decisive Register"), enables reasoning about hyperproperties of programs that use `ABD` via a simpler *shared-memory specification.* (We also use `DR` to demonstrate that multi-writer `ABD` is decisively linearizable by showing a forward simulation to `DR`.) Intuitively speaking, unlike strong linearizability and write strong-linearizability, decisive linearizability gradually commits on the relative order of operations in the sequential history, rather than on the exact position of operations in that history.

Program P_1		Program P_2		Program P_3	
`write(1);`	$b \leftarrow$ `read();`	`write(1);`	`write(2);`	`write(1);`	`write(2);`
`write(2);`		$a \leftarrow$ `coin();`	`barrier();`	`barrier();`	`barrier();`
$a \leftarrow$ `coin();`		`barrier();`	$b \leftarrow$ `read();`	$a \leftarrow$ `coin();`	$b \leftarrow$ `read();`

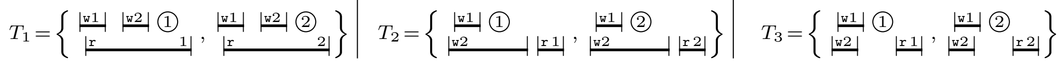


Figure 1 Client programs (upper part) and corresponding trace sets (lower part) that, if an adversary can generate them, violate the hyperproperty "$a = b$ with probability $\frac{1}{2}$"

Outline. The rest of this paper is structured as follows. In §2 we make the introductory discussion more concrete by outlining several examples. In §3 we provide the necessary preliminaries for our formal development. In §4 we introduce and study the notion of a complete implementation for a given linearizability class. In §5 we present the complete implementation for write strong-linearizability. In §6 we define decisive linearizability and present a complete implementation for this class. We discuss related work and conclude in §7. In Appendix A we provide proof sketches for several lemmas and theorems. The full version of this paper [21] provides full proofs.

2 Motivating Examples: A Tale of Four Registers

This section demonstrates certain intricacies arising when examining hyperproperties of client programs using (linearizable) implementations of concurrent registers. The specifications we develop in the next sections are based on the observations arising from these examples. We keep the discussion informal, deferring the formal treatment to the next sections.

Figure 1 (upper part) presents three programs, in which two threads read and write from a shared register, and invoke a method `coin()` that returns 1 or 2 uniformly at random. Programs P_2 and P_3 also employ a synchronization method `barrier()` that ensures the threads wait for each other before executing the rest of the code. Given that the underlying register implementation is linearizable, one can analyze standard properties of a single (finite) trace (e.g., the final values of the variables) of these programs by considering an *atomic* register (`ATR` in Fig. 2) [13]. In technical terms, one says that every linearizable implementation *observationally refines* the atomic register, and that the atomic register provides a *specification* (a.k.a. *reference implementation*) for any linearizable implementation.

However, as observed by Golab et al. [14], the atomic register cannot be used for analyzing properties of *sets* of program traces, a.k.a. *hyperproperties*, which cannot be deduced from a single program trace. For investigating hyperproperties, one considers the sets of program traces that can be generated by an adversary that controls the scheduling and the steps of the implementation. (By *program trace* we mean the sequence of actions performed by the client where the object's implementation internal actions are invisible.) Specifically, we consider the standard *strong* adversary that sees the whole execution so far and makes choices that depend on previous coin-toss results.

For instance, for the programs above, we may aim to verify that *under any adversarial scheduling the probability that $a = b$ at the end of execution is exactly $\frac{1}{2}$*, which indicates

Atomic (ATR)

```
Method read()
    $out \leftarrow X$;
    return $out$;

Method write($v$)
    $X \leftarrow v$;
    return;
```

Double load (DLR)

```
Method read()
    $out_1 \leftarrow X$;
    $out_2 \leftarrow X$;
    if * then return $out_1$;
    else return $out_2$;

Method write($v$)
    $X \leftarrow v$;
    return;
```

Try-not-to-store (TNSR)

```
Method read()
    $out \leftarrow X$;
    return $out$;
Method write($v$)
    $a_1 \leftarrow X$;
    $a_2 \leftarrow X$;
    if * then
        if $a_1 \neq a_2$ then return;
    $X \leftarrow v$;
    return;
```

ABD implementation for N processes ($\texttt{ABD}_N$)

Shared Variables: A set *Broadcasts* of query/update messages and a mapping *Replies* from messages to their replies.
Local Variables: Process p stores the most recent value it observed, v_p, and its timestamp, ts_p. Timestamps are pairs $ts = \langle t, p\rangle$ with $t \in \mathbb{N}$ ordered lexicographically (assuming an arbitrary order on process id's). $\max\{\langle v_1, ts_1\rangle, \ldots, \langle v_n, ts_n\rangle\}$ retrieves the timestamped value $\langle v_i, ts_i\rangle$ with the maximum timestamp.

```
Method read()
    $\langle v, ts\rangle \leftarrow$query();
    update($v, ts$);
    return $v$;

Method write($v$)
    $\langle \_, \langle t, \_\rangle\rangle \leftarrow$query();
    update($v, \langle t+1,$ my_process_id()$\rangle$);
    return;

Function update($v, ts$)
    broadcast $m =$ update($v, ts$);
    wait until $|Replies(m)| > N/2$;
    return;

Function query()
    broadcast $m =$ query;
    wait until $|Replies(m)| > N/2$;
    $Q \leftarrow$pick $Q \subseteq Replies(m)$ s.t. $|Q| > N/2$;
    return $\max Q$;

Background activity by process $p$:
when $m \in Broadcasts$ received
    if $m =$ query then
        reply $\langle v_p, ts_p\rangle$ to $m$;
    if $m =$ update($v, ts$) then
        $\langle v_p, ts_p\rangle \leftarrow \max\{\langle v, ts\rangle, \langle v_p, ts_p\rangle\}$;
        reply "ack" to $m$;
```

Figure 2 Four register implementations

that the adversary cannot leak the coin-toss result from one thread to another.[1] With an atomic register, this property holds in all three programs. For instance, in programs P_1 and P_3, if the adversary performs the atomic `read()` before the coin is tossed, it cannot force a correlation between the coin and the read value; and by the time the coin is tossed, there is only one possible value that can be read.

Next, we demonstrate that this does not mean that other linearizable implementations guarantee this hyperproperty. To this end, we depict below each program in Fig. 1 a set of traces that forces $a = b$ with probability 1, and to show that the hyperproperty of a program is violated for certain implementations, we describe an adversary that generates this set.

We consider three linearizable register implementations, in addition to the atomic register (ATR) discussed above, presented in Figure 2: a "double load" implementation (DLR), a "try-not-to-store" implementation (TNSR), and the well-studied ABD implementation. Like ATR, DLR and TNSR are shared-memory implementations, using a single primitive (atomic) shared memory cell X initialized to 0 (all other variables are local). We refer to the accesses to X as loads/stores, and to the methods of the register as reads/writes. In contrast, ABD is a register implementation in a crash-resilient message passing system, originally proposed

[1] By adding conditional loops in the programs, one can correlate the probability that $a = b$ with the probability that the program diverges, and thus concentrate on asking whether an adversary can force non-termination, as considered in some previous work [15, 6].

to demonstrate that such a system can emulate a shared memory [2]. We present the multi-writer version of `ABD` from [19].

`DLR`. This implementation loads twice and non-deterministically picks which value to return (using **if** $*$). Using `DLR`, in P_1 the adversary can generate T_1 by ensuring this particular interleaving of the two threads, and moreover: execute the first load in the read method after 1 is stored to X, so that $out_1 = 1$; execute the second load after 2 is stored to X, so that $out_2 = 2$; and resolve the non-deterministic choice only after the coin is tossed ensuring that out_1 is returned if the coin result is 1, and out_2 is returned if the coin result is 2. (Recall that the adversary controls object-implementation-internal steps, including non-deterministic choices.) However, it is easy to see that for programs P_2 and P_3, the hyperproperty holds when `DLR` is used. Indeed, without a read concurrently executed with a write, `DLR` behaves just like `ATR`.

`TNSR`. This implementation tries to avoid some stores by recognizing that if the value is concurrently altered during a write operation, then that operation does not have to actually store as it may pretend it was overrun by the concurrent write. With this implementation, the hyperproperty holds for P_1. Indeed, without two concurrently executed writes, `TNSR` behaves just like `ATR`. However, using `TNSR`, in P_2 the adversary can generate T_2 by ensuring that the first load in `write(2)` reads 0 (the initial value), then execute `write(1)` atomically and have the second load in `write(2)` read 1. Then, if the coin result is 1, the adversary makes `write(2)` skip writing its value (it can do so since the two loaded values are not equal). Otherwise, if the coin result is 2, `write(2)` stores its value. Finally, it is easy to see that for P_3 the hyperproperty holds with `TNSR`. Indeed, after both threads reach the barrier, only one value can be returned by the read method, since at this point in the execution, both write methods are completed.

`ABD`. This implementation uses *timestamps* to order the written values (breaking ties using some predetermined order on the process identifiers). Each process maintains the most recent timestamped value it observed. For reading, a process broadcasts a query, waits for replies from a quorum (majority) of processes, and returns the value with the largest timestamp, but only after broadcasting this timestamped value and receiving acknowledgments from a quorum of processes. In turn, for writing value v a process broadcasts a query, waits for replies from a quorum of processes, broadcasts v with timestamp larger than all replies, and waits for a quorum of acknowledgments. Note that in `ABD`, processes are also constantly active as "servers": (*i*) replying to queries with their current timestamped values, and (*ii*) acknowledging new written values after (possibly) updating their current timestamped values.

Using `ABD`, the hyperproperty is violated for P_1 and P_2. For the violation we need to have at least three processes, two of them running the code of the program, and the others are used as servers that reply to messages and participate in quorums. ($\texttt{ABD}_2$ is degenerate since a quorum must consist of all processes.) Essentially, $\texttt{ABD}_{\geq 3}$ allows both the behaviors exposed by `DLR` and the behaviors exposed by `TNSR`. However, the actual adversaries for $\texttt{ABD}_{\geq 3}$ are more complicated than the ones for `DLR` and `TNSR` due to the absence of a global centralized memory cell that values are stored in and loaded from.

We describe adversaries that generate T_1 for P_1 and T_2 for P_2:

- For P_1, the adversary lets the reader invoke a query and lets the writer complete the execution of `write(1)` by choosing a quorum of processes that acknowledge the new value. Next, the adversary lets all the processes in the quorum reply to the query of the reader reporting value 1. Then, the adversary lets the writer execute `write(2)`, again obtaining a quorum of processes that are aware of the new value, where this time

	ATR	DLR	TNSR	$\mathtt{ABD}_{\geq 3}$
P_1	✓	✗	✓	✗
P_2	✓	✓	✗	✗
P_3	✓	✓	✓	✓

For each program and register implementation, ✓ indicates that the hyperproperty "$a = b$ with probability $\frac{1}{2}$" holds under any adversary, and ✗ indicates the hyperproperty is refuted by some adversary.

Figure 3 Summary of examples

the adversary picks a quorum that includes at least one process that is not part of the previous quorum, and therefore has not yet replied to the reader's query (this is where at least three processes are needed). This process also replies to the reader's query, but with value 2. At this time, the query message of the reader has pending replies from a quorum in which all replies include value 1, and from one additional process that is already aware of the more recent value 2. However, the adversary postpones the delivery of the replies until after the coin toss, at which time it picks the replies to match the coin value: if the coin value is 1, the replies from the homogeneous quorum where all replies include value 1 are delivered; otherwise the replies from the all but of one of the processes in the aforementioned quorum are delivered together with the reply of the additional process that includes value 2, thus forming a (heterogeneous) quorum whose most recent value is 2. Accordingly, the reader returns a value that is equal to the coin value.

- For P_2, the adversary starts by invoking a query during `write(2)` and making a quorum of processes send replies to the query (with the initial value) before `write(1)` is initiated in the left process. The adversary then lets the left process execute up to the barrier, at which point at least one reply with the value 1 is sent to the right process's query by a process that is aware of the left process's update. The adversary then performs the delivery of the replies to the query in `write(2)` according to the coin value. If the coin result is 2, the adversary delivers a quorum of replies that includes the reply sent when the left process reached the barrier, causing the right process to be aware of the most recent timestamp of the left process, such that the right process updates the value 2 with a larger timestamp. On the other hand, if the coin result is 1, the adversary delivers only the replies sent before `write(1)`, whose timestamp is outdated, causing the right process to choose a timestamp for the new value 2 that is at a tie with the timestamp attached by the left process to the value 1. Assuming the id of the left process has precedence, the tie is resolved to its timestamp, making 1 appear to be the most recent value. This determines the result of the subsequent read to be equal to the coin result.

Finally, the hyperproperty holds when ABD is used in P_3. To see this, suppose, w.l.o.g., that the timestamp assigned to 2 is larger than the one of 1. Then, after the two writes complete, in every quorum there is at least one process that knows about the value 2, and a reader that queries after this point can only read 2.

Figure 3 summarizes the above observations. In particular, the hyperproperty holds in P_3 for all four implementations. Nevertheless, as we show later in Example 6.2, it can be still violated by some linearizable implementations.

To capture differences between linearizable implementations, such as the ones shown in the above examples, [3] introduced *strong observational refinement* as a refinement relation between an implementation and a specification that preserves hyperproperties. Then, while ATR, DLR, TNSR, and ABD can be shown to be observationally equivalent (i.e., observationally refine each other), as we demonstrated above, they are not *strongly* observational equivalent. In particular, none of the relatively simple shared-memory implementations in Fig. 2 can be used as a specification of ABD when hyperproperties are considered, as ABD is not a strong observational refinement of any of them. (This is unfortunate, since, as we have seen, reasoning about the sets of program traces generated when ABD is used is much more involved than with the other implementations.) We also note that each of the implementations admits a different linearizability criterion: ATR is strongly linearizable [14], DLR is write strongly-linearizable [15], while TNSR and ABD are neither.

In the rest of the paper we propose hyperproperty-preserving specifications for classes of linearizable register implementations, including ABD. Such specifications can drastically simplify verification of hyperproperties of client programs using these implementations, a task which is typically challenging, especially when complex implementations are considered, since it requires reasoning about *all* possible adversaries.

3 Preliminaries

We start with general notations, continue to our modeling of objects, implementations, and programs (§3.1), and finally recap the formal notions of preservation of hyperproperties via strong observational refinement (§3.2).

Sequences. For a finite alphabet Σ, we denote by Σ^* the set of all (finite) sequences over Σ. The length of a sequence s is denoted by $|s|$. We write $s[k]$ for the symbol at position $1 \leq k \leq |s|$ in s. We write $\sigma \in s$ if $s[k] = \sigma$ for some $1 \leq k \leq |s|$. We use "$\cdot$" for the concatenation of sequences. We often identify symbols with sequences of length 1 or their singletons (e.g., in expressions like $s \cdot \sigma$). The *restriction* of a sequence s w.r.t. a set Γ, denoted by $s|_\Gamma$, is the longest subsequence of s that consists only of symbols in Γ. This notation is extended to sets by $S|_\Gamma \triangleq \{s|_\Gamma \mid s \in S\}$. We write $s_1 \preceq_{\mathsf{S}} s_2$ when s_1 is a subsequence of s_2, and $s_1 \preceq_{\mathsf{P}} s_2$ when s_1 is a prefix of s_2.

Labeled Transition Systems. A *labeled transition system* (LTS, for short) is a tuple $A = \langle Q, \Sigma, q_0, T\rangle$, where Q is a set of *states*, Σ is a (possibly infinite) alphabet (whose elements are called *transition labels*), $q_0 \in Q$ is an *initial state*, and $T \subseteq Q \times \Sigma \times Q$ is a set of *transitions*. We denote by $A.\mathtt{Q}$, $A.\boldsymbol{\Sigma}$, $A.\mathtt{q}_0$, and $A.\mathtt{T}$ the components of an LTS A. We write $\xrightarrow{\sigma}_A$ for the relation $\{\langle q, q'\rangle \mid \langle q, \sigma, q'\rangle \in A.\mathtt{T}\}$. An *execution* e of A is a (possibly empty) finite sequence of transitions in $A.\mathtt{T}$ such that the first transition starts in q_0 and each other transition continues from the target of the previous transition. An execution e induces a *trace* $\rho \in A.\boldsymbol{\Sigma}^*$, where $\rho[i]$ is given by the label of $e[i]$ for every $1 \leq i \leq |e|$. We denote by $\mathsf{E}(A)$ and $\mathsf{traces}(A)$ the set of all executions of A and the set of all traces induced by executions of A (respectively). Note that we only consider finite executions and traces.

Forward Simulations. Given LTSs A and $A^\#$ and a set $\Gamma \subseteq A.\boldsymbol{\Sigma} \cap A^\#.\boldsymbol{\Sigma}$, a relation $R \subseteq A.\mathtt{Q} \times A^\#.\mathtt{Q}$ is a Γ-*forward simulation* from A to $A^\#$ if (i) $\langle A.\mathtt{q}_0, A^\#.\mathtt{q}_0\rangle \in R$; and (ii) if $q \xrightarrow{\sigma}_A q'$ and $\langle q, q^\#\rangle \in R$, then there exist $q^{\#'} \in A^\#.\mathtt{Q}$ and $\rho \in A^\#.\boldsymbol{\Sigma}^*$ such that $q^\# \xrightarrow{\rho[1]}_{A^\#} \dots \xrightarrow{\rho[|\rho|]}_{A^\#} q^{\#'}$, $\rho|_\Gamma = \sigma|_\Gamma$, and $\langle q', q^{\#'}\rangle \in R$. We write $A \sqsubseteq^\Gamma_{\mathsf{F}} A^\#$ when such relation exists.

3.1 Objects, Implementations, and Programs

We review standard notions that are needed for our formal results. We assume a set Tid of thread identifiers and an infinite set Id of action identifiers.

Objects. An *object* is a pair $O = \langle \mathsf{M}, \mathsf{Val}\rangle$, where M is a set of method names and Val is a set of values. An object O is associated with actions divided into *invocations* $i = \mathtt{inv}\langle m, v, p, k\rangle \in$

$\mathsf{I}(O)$ and *responses* $r = \mathtt{res}\langle m, v, p, k\rangle \in \mathsf{R}(O)$, where $m \in \mathsf{M}$, $v \in \mathsf{Val} \cup \{\bot\}$, $p \in \mathsf{Tid}$, and $k \in \mathsf{Id}$. We let $\mathsf{IR}(O) \triangleq \mathsf{I}(O) \cup \mathsf{R}(O)$.

Histories. A *history* h of an object O is a finite sequence over $\mathsf{IR}(O)$. A history h is *sequential* if it alternates between invocations and responses (starting with an invocation), such that every consecutive i, r in h have the same method and thread identifiers, and a unique action identifier across h. A history h is *well-formed* if its restriction to actions of each $p \in \mathsf{Tid}$, denoted by $h|_p$, is sequential. An invocation $i \in h$ is *pending* if there is no response in h with the same thread and action identifiers. Otherwise, i is *complete*. These notions are also applied on *operations* o, which are either single invocations $o = i$ or pairs of matching invocation and response $o = \langle i, r\rangle$. We let $\mathsf{completed}(h)$ denote the subsequence of h consisting of actions that are a part of completed operations.

Real-time Order. The *real time order* induced by a well-formed history h, denoted by $<_h$, is the partial order on operations defined by $o_1 <_h o_2$ iff o_1's response appears in h before o_2's invocation.

Specifications. A *specification* of O is a prefix-closed set of sequential histories of O.

Registers. A register object is given by $\mathsf{Reg} = \langle\{\mathtt{read}, \mathtt{write}\}, \mathbb{N}\rangle$. Its specification, denoted by $\mathsf{Spec}_{\mathsf{Reg}}$, is defined as usual, assuming that 0 is the initial register value.

Object Implementations. We assume a set IInt of labels for implementation internal actions and define an *implementation* I of an object O to be an LTS over the alphabet $\mathsf{IR}(O) \cup \mathsf{IInt}$. We assume that the history induced by every execution e of I, denoted by $\mathsf{h}(e)$, is a well-formed history. The pseudo-code presented in specific implementations in the paper is easily translatable to formal LTSs, whose executions represent executions generated by the methods' code when they are repeatedly and concurrently invoked with arbitrary arguments.

Client Programs. We assume a set PInt of labels for client internal actions (disjoint from IInt) and define a client program P for an object O as an LTS over the alphabet $\mathsf{IR}(O) \cup \mathsf{PInt}$. A program P and implementation I are *linked* by taking "interface parallel composition", denoted by $P[I]$. The resulting LTS interleaves the steps of P and I while forcing the two LTSs to synchronize on labels from $\mathsf{IR}(O)$. The defining property of $P[I]$ is given by:

▶ **Proposition 3.1.** $\rho \in \mathsf{traces}(P[I])$ *iff* $\rho|_{I.\mathbf{\Sigma}} \in \mathsf{traces}(I)$ *and* $\rho|_{P.\mathbf{\Sigma}} \in \mathsf{traces}(P)$.

3.2 Hyperproperties Preservation via Strong Observational Refinement

A *hyperproperty* ϕ of a program P is a set of sets of the program's traces (i.e., $\phi \subseteq \mathcal{P}(\mathsf{traces}(P))$). Such sets can capture probabilistic requirements, such as the one informally described in §2, via suitable encodings of traces [9].

The hyperproperties that are satisfied by an object implementation, and accordingly, strong observational refinement between implementations, are defined using deterministic schedulers, which formalize the notion of a strong adversary [3].

Schedulers. Given a program P and an implementation I, a *scheduler* is a function $S : \mathsf{E}(P[I]) \to \mathcal{P}(P[I].\mathtt{T})$. An execution $e \in \mathsf{E}(P[I])$ is *consistent with* S if $e[j] \in S(e[1] \cdots e[j-1])$ for every $1 \leq j \leq |e|$. We denote by $\mathsf{E}(P[I], S)$ the set of executions of $P[I]$ that are consistent with S, and by $\mathsf{traces}(P[I], S)$ the traces of executions in $\mathsf{E}(P[I], S)$. A scheduler is *deterministic* if for every $e \in \mathsf{E}(P[I])$, either $|S(e)| \leq 1$ or all transitions in $S(e)$ are labeled by actions in PInt.

▶ Remark 3.2. Attiya and Enea [3] restricted their attention to *step-deterministic* implementations in which a trace uniquely determines an execution (which includes the intermediate states along the trace). We avoid this technical restriction, and thus use executions instead of traces in the definitions of schedulers, as well as of linearizability criteria below. In particular, we define schedulers as functions from executions to sets of transitions instead of functions from traces to sets of labels. For step-deterministic implementations our definitions coincide with those of [3].

Hyperproperty Satisfaction. An implementation I *satisfies a hyperproperty* ϕ of P, denoted by $I \models_P \phi$, if $\mathsf{traces}(P[I], S)|_{P.\mathbf{\Sigma}} \in \phi$ for every deterministic scheduler S.

▶ **Example 3.3.** For the client program P_2 (represented as an LTS) and the set of traces T_2 from Fig. 1, we have that $\mathtt{DLR} \models_{P_2} \mathcal{P}(\mathsf{traces}(P_2)) \setminus \{T_2\}$. This is because, as discussed in §2, there exists no scheduler S such that $\mathsf{traces}(P_2[\mathtt{DLR}], S)|_{P.\mathbf{\Sigma}} = T_2$. ⌟

Strong Observational Refinement. An implementation I *strongly observationally refines* an implementation $I^\#$, denoted by $I \leq_\mathsf{s} I^\#$, if $I^\# \models_P \phi \implies I \models_P \phi$ for every program P and hyperproperty ϕ of P. The following alternative characterization follows from the definition.

▶ **Lemma 3.4.** *$I \leq_\mathsf{s} I^\#$ iff for every program P and deterministic scheduler S, there exists a deterministic scheduler $S^\#$ such that* $\mathsf{traces}(P[I], S)|_{P.\mathbf{\Sigma}} = \mathsf{traces}(P[I^\#], S^\#)|_{P.\mathbf{\Sigma}}$.

Attiya and Enea [3, Theorem 8] show that $\mathsf{IR}(O)$-forward simulation between implementations is equivalent to strong observational refinement. (Their result applies to finite traces as we consider here; see [12] for a discussion on infinite traces.) We adapt this result to our setting. In the sequel, for implementations I and $I^\#$ of an object O, we write $I \sqsubseteq_\mathsf{F} I^\#$ for $I \sqsubseteq_\mathsf{F}^{\mathsf{IR}(O)} I^\#$.

▶ **Theorem 3.5.** *$I \leq_\mathsf{s} I^\#$ iff $I \sqsubseteq_\mathsf{F} I^\#$.*

▶ **Example 3.6.** It is easy to show that $\mathtt{ATR} \sqsubseteq_\mathsf{F} \mathtt{DLR}$, and we obtain that $\mathtt{ATR} \leq_\mathsf{s} \mathtt{DLR}$. Thus, $\mathtt{ATR} \models_{P_2} \mathcal{P}(\mathsf{traces}(P_2)) \setminus \{T_2\}$ follows from $\mathtt{DLR} \models_{P_2} \mathcal{P}(\mathsf{traces}(P_2)) \setminus \{T_2\}$. In addition, since $\mathtt{DLR} \not\leq_\mathsf{s} \mathtt{ATR}$ (see §2), we have $\mathtt{DLR} \not\sqsubseteq_\mathsf{F} \mathtt{ATR}$. Indeed, if a concurrent write is about to change the value of X after a read of DLR performs its first load, ATR has no matching action: if it performs its (single) load it will not be able to return the right value in case DLR returns the value read in the second load; and similarly, if it waits, it will fail to return the same value if DLR returns the value of the first load. ⌟

4 Complete Implementations for Linearizability Classes

Knowing that a given implementation is a member of a certain linearizability class is only useful if it enables reasoning about programs that use that implementation without understanding the implementation itself. For hyperproperties, such reasoning is made possible if the implementation is known to strongly observationally refine a simpler implementation, in which case the latter can be used instead of the actual implementation in the analysis. To standardize the relation between linearizability classes and strong observational refinement, we propose a definition of *hard* and *complete* implementations in analogy to hardness and completeness w.r.t. complexity classes, where instead of reductions, we use simulations, which ensure strong observational refinement:

▶ **Definition 4.1.** Let $\mathcal{I}$ be a class of implementations of an object O that is downward closed w.r.t. forward simulation (i.e., $I \in \mathcal{I}$ whenever $I' \in \mathcal{I}$ and $I \sqsubseteq_\mathsf{F} I'$). An implementation $I^\#$ of O is *$\mathcal{I}$-hard* if $I \sqsubseteq_\mathsf{F} I^\#$ for every $I \in \mathcal{I}$. It is *$\mathcal{I}$-complete* (or *complete for $\mathcal{I}$*) if we also have $I^\# \in \mathcal{I}$.

In addition to allowing reasoning about hyperproperties of implementations in $\mathcal{I}$, an $\mathcal{I}$-complete implementation $I^{\#}$ also provides a sound and complete method to establish the membership of an implementation I in $\mathcal{I}$ by showing that $I \sqsubseteq_{\mathsf{F}} I^{\#}$.

In the following we take $\mathcal{I}$ to be the set of implementations of some object that satisfy certain linearizability criteria.

Linearizability. Consider first standard linearizability [17, 20]:

▶ **Definition 4.2.** A history s of an object O is a *linearization* of a history h of O, denoted by $h \sqsubseteq s$, if there exists a sequence of responses $\bar{r}$ for some of the pending invocations in h such that the following hold for $h' = \mathsf{completed}(h \cdot \bar{r})$: (i) $h'|_p = s|_p$ for every $p \in \mathsf{Tid}$; and (ii) $<_{h'} \subseteq <_s$. A history h of O is *linearizable* w.r.t. a specification *Spec* of O if it has a linearization $s \in \mathit{Spec}$. An implementation I of O is *linearizable* w.r.t. *Spec* if $\mathsf{h}(e)$ is linearizable w.r.t. *Spec* for every $e \in \mathsf{E}(I)$.

▶ **Proposition 4.3.** *The class of linearizable implementations of an abject O w.r.t. a specification Spec is downward closed w.r.t. forward simulation, and there exists a complete implementation for it.*

Proof (sketch). Downward closedness follows from the fact that $I \sqsubseteq_{\mathsf{F}} I'$ implies that $\{\mathsf{h}(e) \mid e \in \mathsf{E}(I)\} \subseteq \{\mathsf{h}(e) \mid e \in \mathsf{E}(I')\}$. A complete implementation is the implementation that tracks in its internal state the history h generated so far. When executing an invocation or response, the action is added in the end of the current history. But, while invocations are always enabled, a response r is only enabled when $h \cdot r$ is linearizable w.r.t. *Spec*. ◀

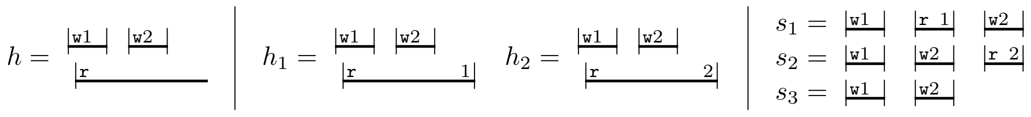
The (theoretical) construction in the above proof provides us with a complete implementation, which may help in streamlining and mechanizing linearizability arguments as forward simulations (e.g., [18] utilized such implementation). However, since it directly encodes the definition of the class, it is unhelpful for reasoning about hyperproperties of implementations. Thus, for the stronger classes considered below we are interested in identifying simple complete implementations that are not based on history tracking.

Strong linearizability. Golab et al. [14] proposed a strengthening of linearizability, called *strong linearizability*, and showed that it is necessary and sufficient for reasoning on probability distributions of outcomes that a strong adversary can generate. Roughly speaking, while linearizability allows one to choose the linearization order "after the fact" in view of the whole execution, strong linearizability requires the linearization of implementation histories into specification histories to be done online in a prefix-preserving manner, that is, by continuously adding operations at the end of the linearized history.

▶ **Definition 4.4.** A *linearization mapping* for an implementation I of an object O w.r.t. a specification *Spec* of O is a function $L : \mathsf{E}(I) \to \mathit{Spec}$ such that $\mathsf{h}(e) \sqsubseteq L(e)$ for every $e \in \mathsf{E}(I)$. An implementation I of O is *strongly linearizable* w.r.t. a specification *Spec* of O if there is a linearization mapping L for I w.r.t. *Spec* such that $L(e_1) \preceq_{\mathsf{P}} L(e_2)$ whenever $e_1 \preceq_{\mathsf{P}} e_2$.

Since we aim to also capture non-deterministic implementations (and do not assume step-determinism), our linearizations apply on *executions* rather than traces (see also Remark 3.2).

▶ **Example 4.5.** From the register implementations presented in §2, only `ATR` is strongly linearizable. We use the histories associated with the set T_1 from Fig. 1 to show that `DLR` and `ABD` are not strongly linearizable. Consider the following history h, its two possible extensions h_1 and h_2, and its possible linearizations s_1, s_2, s_3:



Unlike ATR (and TNSR), both DLR and ABD have a *single* execution e that induces h and can be extended into two alternative executions that induce h_1 and h_2. Then, $L(e)$ can be s_1, s_2, or s_3, but any choice at this stage is doomed to fail: (i) s_1 fails if the execution continues to generate h_2; (ii) s_2 fails if the execution continues to generate h_1; and (iii) s_3 fails if the execution continues to generate h_1 since we are only allowed to extend the current linearization by adding operations at its end. The history of the common prefix of the traces in T_2 from Fig. 1 can be similarly used to show that TNSR is not strongly linearizable. ⌟

Attiya and Enea [3] show that the class of strongly linearizable implementations is downward closed w.r.t. forward simulation, and that every strongly linearizable implementation strongly observationally refines the atomic implementation (e.g., ATR for registers). Together with Thm. 3.5, this result is restated as follows:

▶ **Theorem 4.6.** *The atomic implementation for specification Spec of an object O is complete for the class of strongly linearizable implementations of O w.r.t. Spec.*

Additional linearizability classes. We observe that downward-closedness w.r.t. simulation, as well as the existence of a complete implementation, generalize to a range of linearizability classes beyond linearizability and strong linearizability mentioned above. These linearizability classes are parameterized by a preorder that must hold between the linearizations of an execution and its extensions. Formally, given a preorder R (i.e., reflexive and transitive relation) on sequences, the class $\mathcal{I}_R(O, \mathit{Spec})$ consists of all implementations I of O for which there exists a linearization mapping $L : \mathsf{E}(I) \to \mathit{Spec}$ such that $\langle L(e_1), L(e_2)\rangle \in R$ whenever $e_1 \preceq_{\mathsf{P}} e_2$. The class of all linearizable implementations of O w.r.t. *Spec* is obtained by taking $R = \mathit{Spec} \times \mathit{Spec}$, whereas for all strongly linearizable implementations we take $R = \preceq_{\mathsf{P}}$. Other classes defined in the rest of this paper are also instances of this definition.

▶ **Lemma 4.7.** *For every preorder R on sequences, the class $\mathcal{I}_R(O, \mathit{Spec})$ is downward closed w.r.t. forward simulation, and there exists a complete implementation for it.*

The complete implementation for $\mathcal{I}_R(O, \mathit{Spec})$ is constructed similarly to the one in the proof of Prop. 4.3 (which is a special case), except that here the state also tracks a linearization of the history so far, and ensures in each transition that the linearizations in the pre-state and post-state are related by R.

Similarly to the construction in Prop. 4.3, the generic construction in Lemma 4.7 is not helpful for reasoning about hyperproperties. In contrast, Thm. 4.6 proposes a simple and useful complete implementation for strong linearizability. In the remainder of the paper we seek useful complete implementations for other linearizability classes of interest.

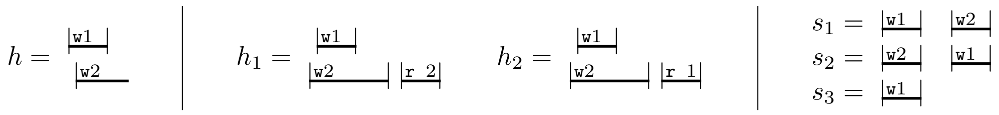
5 Complete Implementation for Write Strong Linearizability

Focusing on registers and identifying that useful register implementations are not strongly linearizable, Hadzilacos et al. [15] have recently proposed a weakening of strong linearizability, called *write strong-linearizability*, and showed that every linearizable *single writer* register implementation, including single-writer ABD, is write strongly-linearizable. However, they do not provide a specification for write strong-linearizability that plays the role that the atomic register implementation plays for strong linearizability.

Write strong-linearizability weakens the prefix-preservation requirement of strong linearizability by applying it only to writes, thus allowing reads to be linearized offline, and freely "move around" when more operations are added. For the formal definition, we let $s|_{\texttt{write}}$ denote the restriction of $s \in \mathsf{Spec}_{\mathsf{Reg}}$ to write operations.

▶ **Definition 5.1.** Let I be a register implementation. A linearization mapping $L : \mathsf{E}(I) \to \mathsf{Spec}_{\mathsf{Reg}}$ is *write strong* if $L(e_1)|_{\texttt{write}} \preceq_{\mathsf{P}} L(e_2)|_{\texttt{write}}$ whenever $e_1 \preceq_{\mathsf{P}} e_2$. We say that I is *write strongly-linearizable* if there exists a write strong linearization mapping $L : \mathsf{E}(I) \to \mathsf{Spec}_{\mathsf{Reg}}$.

▶ **Example 5.2.** From the implementations in §2, `ATR` and `DLR` are write strongly-linearizable. (For `DLR`, for h from Example 4.5, we can pick s_3, and later on, when the read returns, pick either s_1, by adding a read in the middle, or s_2 according to the returned value.) We use the histories associated with the set T_2 from Fig. 1 to show that `TNSR` and `ABD` are not write strongly-linearizable. (For `ABD` this also follows from the general result in [8].) Consider the following history h, its two possible extensions h_1 and h_2, and its possible linearizations s_1, s_2, s_3:



Unlike `ATR` and `DLR`, both `TNSR` and `ABD` have a *single* execution e that induces h and can be extended into two alternative executions that induce h_1 or h_2. Then, $L(e)$ can be s_1, s_2, or s_3, but any choice at this stage is doomed to fail: (i) s_1 fails if the execution continues to generate h_2 since no extension of s_1 linearizes h_2; (ii) s_2 fails if the execution continues to generate h_1 since no extension of s_2 linearizes h_1; and (iii) s_3 fails if the execution continues to generate h_2 since no extension of s_3, where write operations are only added after the write operation in s_3, linearizes h_1. ⌟

We denote by $\mathcal{I}_{\mathsf{ws}}$ the class of write strongly-linearizable register implementations. By Lemma 4.7 (with R ordering histories using the prefix relation on the restriction to writes), $\mathcal{I}_{\mathsf{ws}}$ is downward-closed w.r.t. simulation, and the notion of a complete implementation is well-defined. Algorithm 1 presents our proposed complete implementation for this class. Its construction is inspired by a specification given by Attiya and Enea [3, §6] for capturing the hyperproperties of a specific snapshot implementation [1]. It is a generalization of `DLR` from §2, where instead of loading twice, the reader repeatedly loads from X as long as new values are observed, and non-deterministically decides which value to return.

▶ Remark 5.3. One can define a sequence $\{I_k\}_{k=1}^{\infty}$ of implementations, all with atomic write, and read that non-deterministically picks between k-loads (so `ATR` $= I_1$ and `DLR` $= I_2$). It can be shown that all of these implementations are write strongly-linearizable, but for every k, I_{k+1} does not strongly observationally refine I_k. The `WSR` implementation is what one gets "at the limit" of this sequence, and every I_k trivially strongly observationally refines `WSR`.

▶ **Theorem 5.4.** `WSR` *is complete for the class of write strongly-linearizable register implementations.*

As a consequence of Thm. 5.4, we obtain that single-writer `ABD` strongly observationally refines `WSR`, and so we can use `WSR` to argue about the hyperproperties of client programs that use single-writer `ABD`.

Algorithm 1 WSR: A complete implementation for write strongly-linearizable registers

```
Shared Variables: the current value $X$.
Multi-assignments are executed atomically.
Method read()                                  Method write($v$)
    $\mathcal{V} \leftarrow \{X\}$;                      $X \leftarrow v$;
    do                                             return;
        $\langle \mathcal{V}_{\text{prev}}, \mathcal{V} \rangle \leftarrow \langle \mathcal{V}, \mathcal{V} \cup \{X\} \rangle$;
    while $\mathcal{V} \neq \mathcal{V}_{\text{prev}}$;
    $out \leftarrow$ pick $v \in \mathcal{V}$;
    return $out$;
```

6 Complete Implementation for Decisive Linearizability

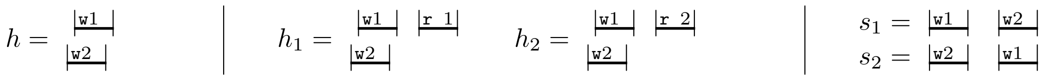
In this section we identify a novel linearizability criterion, which we call *decisive linearizability*. Then, we present a complete implementation for the corresponding class of register implementations, which can serve as a hyperproperty-preserving specification for any implementation in the class. Using this implementation, we show that multi-writer ABD is decisively linearizable, and that decisive linearizability (for registers) is weaker than write strong-linearizability.

▶ **Definition 6.1.** Let I be an implementation of an object O and *Spec* be a specification of O. A linearization mapping $L : \mathsf{E}(I) \to \mathit{Spec}$ is *decisive* if $L(e_1) \preceq_{\mathsf{S}} L(e_2)$ whenever $e_1 \preceq_{\mathsf{P}} e_2$. We say that I is *decisively linearizable* w.r.t. *Spec* if there there exists a decisive linearization mapping $L : \mathsf{E}(I) \to \mathit{Spec}$.

Decisive linearizability, like strong and write strong-linearizability, requires the linearization process to be "online". Nevertheless, unlike strong and write strong-linearizability, it does not require that the sequences of linearizations produced in this process are increasing "at the end", thus allowing operations to be added to the linearized history possibly before operations that are already included in the linearized history. The only requirement of decisive linearizability is that this process maintains the relative order of already linearized operations: once the order between o_1 and o_2 has been decided, it cannot be reverted.

▶ **Example 6.2.** All implementations in §2 are decisively linearizable: ATR and DLR are already write strongly-linearizable (which is a stronger condition, as we show below) and for TNSR and ABD, which are not write strongly-linearizable, this will be proven later in the section. To illustrate how a suitable linearization mapping can be obtained for these implementations, we revisit the histories h and its extensions h_1 and h_2 from Example 5.2. To linearize h, we can pick s_3; later on, if the execution continues according to h_1, we append w2 to the linearization, and if the execution continues to h_2, we add w2 to the linearization before w1—note that decisive linearizability allows this; finally, when the read returns we add it immediately after the corresponding write.

For a "non-example", we use the histories associated with the set T_3 from Fig. 1 to show that the complete implementation for the class of linearizable registers (see Prop. 4.3) is not decisively linearizable. Consider the following history h, its two possible extensions h_1 and h_2, and its possible linearizations s_1 and s_2:



Recall that in the complete implementation for standard linearizability, an execution e that induces h can be extended both to an execution e_1 that induces h_1 and to an execution e_2 that induces h_2. (In particular, this means that an adversary for P_3 from Fig. 1 can

Algorithm 2 `DR`: A complete implementation for decisively linearizable registers

Shared Variables: the current value X, the current version number Ver, and a lock flag L.
await B **do** C blocks until the condition B is met, at which point the evaluation of B and the body C are atomically executed. Multi-assignments and **atomic** blocks are executed atomically.

```
Method read()
    await $L = 0$ do $\langle s, \mathcal{V}\rangle \leftarrow \langle Ver, \{X\}\rangle$;
    do
        atomic
            $\mathcal{V}_{\text{prev}} \leftarrow \mathcal{V}$;
            if $Ver \geq s$ then $\mathcal{V} \leftarrow \mathcal{V} \cup \{X\}$;
    while $\mathcal{V} \neq \mathcal{V}_{\text{prev}}$;
    $out \leftarrow$ pick $v \in \mathcal{V}$;
    return $out$;
```

```
Method write($v$)
    await $L = 0$ do $s \leftarrow Ver$;
    if * then
        await $L = 0$ do $\langle X, Ver\rangle \leftarrow \langle v, Ver + 1\rangle$;
    else
        await $L = 0 \wedge Ver > s$ do
            $\langle L, tmp, X, Ver\rangle \leftarrow \langle 1, X, v, Ver - 1\rangle$;
        $\langle L, X, Ver\rangle \leftarrow \langle 0, tmp, Ver + 1\rangle$;
    return;
```

decide between these options after the coin toss, refuting the hyperproperty discussed in §2, which is satisfied when each of `ATR`, `DLR`, `TNSR`, `ABD` and in fact any decisively linearizable implementation is used.) If $L(e) = s_1$ then the linearization of e_1 must reorder the writes in s_1, violating decisiveness. Similarly, if $L(e) = s_2$, then the linearization of e_2 must reorder the writes in s_2, violating decisiveness. Thus, no decisive linearization mapping exists. ⌟

By Lemma 4.7 (with R being the subsequence relation), the class of decisively linearizable implementations is downward-closed w.r.t. simulation and a complete implementation exists for it, for any object. Next, we present a complete implementation for the class of decisively linearizable register implementations. We note that while Definition 6.1 is not specific to registers (unlike Definition 5.1) and Lemma 4.7 applies to any object, the complete implementation we present is only for register implementations. We denote by $\mathcal{I}_{\mathsf{d}}$ the class of all decisively linearizable register implementations. The complete implementation, `DR`, is presented in Algorithm 2.

`DR` stores the current value in X and a corresponding version number in Ver. Reads use repeated loads similarly to `WSR`, but add loaded values to $\mathcal{V}$ only when their version number is not older than the version number when the read started (stored in s). The return value is picked non-deterministically from $\mathcal{V}$.

Writes are based on the idea used in `TNSR`, allowing stores to non-deterministically choose to be overwritten by a concurrent write, with two important differences. First, new stores by concurrent writes are identified based on version number ($Ver > s$) rather than values (to avoid data dependencies). Second, even if a write chooses to be overwritten, the store to X is not skipped but momentarily executed with a lower version number, to allow concurrent reads to observe it. This is done by a step that temporarily decreases Ver and stores the input value to X, followed by a step that restores Ver and X to their newer values. The two steps are not executed atomically, letting concurrent reads to load the intermediate value. Importantly, a lock L is used to prevent concurrent methods from setting their start version number (s) to a temporary version number, and from updating Ver based on a temporary version number.

To simplify the presentation, the pseudo-code is written such that a write makes the non-deterministic choice whether to be overwritten or not before it determines that it can indeed be overwritten. As a result, the execution may get stuck. This does not affect linearizability, and this behavior is impossible in our formulation of `DR` as an LTS.

▶ **Example 6.3.** Allowing concurrent reads to observe "overwritten" writes is crucial for capturing all behaviors of decisively linearizable implementations such as multi-writer `ABD`. Consider the program P_4 and set of traces T_4 in Fig. 4. The program P_4 extends P_2 from

```
write(1);       ||  write(2);      ||  c ← read();
a ← coin();     ||  barrier();     ||  barrier();
barrier();      ||  b ← read();    ||
```

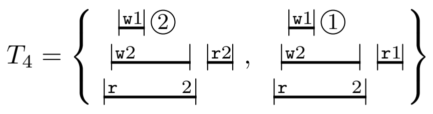


Figure 4 A program P_4 and a set T_4 of traces of the program

Fig. 1 with another thread, and T_4 is similar to T_2 except that the additional thread observes the value 2 written by the middle thread, even when this value ends up being overwritten. Recall that T_2 can be generated by an adversary for both TNSR and ABD. For TNSR, this leverages the ability of the adversary to postpone the decision whether to store 2 or not until after the coin toss. In contrast, T_4 is not possible for TNSR, since in the trace where the middle thread reads 1, it must be the case that TNSR chose to overwrite 2 and as a result has never stored 2 to X, preventing concurrent threads from loading the value before it is overwritten. (DR does perform a store in such a case, allowing T_4.) Unlike TNSR, ABD allows this behavior: The adversary acts on the left and middle processes similarly to the adversary for P_2 that generates T_2 described in §2, with the added right process sending an additional query when write(2) does so, immediately receiving replies with the initial value from a set of process that excludes the middle process and is one-short from a quorum. Then, when write(2) sends its update, it also replies to the right process with the timestamp it chose. Regardless of the chosen timestamp, it is larger than the initial timestamp, causing the right process to return the value 2. ⌟

▶ **Theorem 6.4.** DR *is complete for the class of decisively linearizable register implementations.*

While not immediate from the definitions, a corollary of Theorems 5.4 and 6.4, together with the observation that WSR $\sqsubseteq_{\mathsf{F}}$ DR, is that every write strongly-linearizable implementation is also decisively linearizable. That is, decisive linearizability is indeed weaker than write strong-linearizability.

Having constructed a complete implementation, we now leverage it to show that other implementations are decisively linearizable: all we need to do is prove that they are simulated by DR. For example, TNSR is trivially simulated by DR, and is therefore decisively linearizable. We show that the same holds for multi-writer ABD.

▶ **Theorem 6.5.** ABD $\sqsubseteq_{\mathsf{F}}$ DR.

▶ **Corollary 6.6.** ABD *is decisively linearizable.*

Thus, DR provides a shared-memory specification for multi-writer ABD that enables reasoning about its hyperproperties.

7 Related and Future Work

Since the observation that linearizability does not suffice for reasoning about randomized client programs and the introduction of strong linearizability [14], many works have studied (im)possibility of implementing strongly linearizable objects under different progress conditions. Helmi et al. [16] showed that lock-free strongly linearizable multi-writer registers, max registers, snapshots, and counters cannot be constructed from a single-writer registers. Attiya et al. [5] and Chan et al. [8] adapted and extended these results for a fault-tolerant message passing setting.

Attiya et al. [6] developed a methodology of making existing implementations probabilistically close to strongly linearizable ones by repeating an effect-free preamble of every method and picking uniformly at random which outcome to continue with. They introduced a correctness condition called *tail strong linearizability* that ensures the effectiveness of this construction. This criterion depends on the choice of the preamble and is thus not comparable to decisive linearizability. Interestingly, the construction in [6] is not effective for our complete implementations (WSR and DR).

The work of Hadzilacos et al. [15] is closer to our work in its aim to give up strong linearizability, and study what existing implementations do provide. In addition to what we have already discussed, [15, Algorithm 4] demonstrated a multi-writer register implementation that is not write strongly-linearizable. This implementation is essentially a simplified version of ABD, and using forward simulation to ABD, one can conclude that it is decisively linearizable.

As discussed in length, our work is heavily inspired by [3] that uncovered the correspondence between strong observational refinement and simulation, and suggested the use of non-atomic specifications for reasoning about non-strongly-linearizable implementations. Derrick et al. [10] and Dongol et al. [12] identified a gap in the way [3] handle infinite traces, and show that in that case, while simulation is still necessary for strong observational refinement, only a stronger relation, called *(weak) progressive forward simulation* is sufficient. We focus solely on finite traces, leaving infinite traces to future work.

Bouajjani et al. [7] used forward simulations to non-atomic reference implementations as means to establish linearizability. In particular, they developed abstract stack and queue specifications such that forward simulations to these specifications is necessary and sufficient for establishing linearizability. In our terms, this gets close to complete implementations for the class of linearizable stacks and queues, but, their results are, however, limited to implementations that have explicit marking of linearization points (or so-called "commit points") in some of the methods. Their implementations are highly beneficial in simplifying (and possibly automating) complex linearizability arguments, as the ones needed for Herlihy&Wing Queue [17] and the Time-Stamped Stack [11].

Finally, we note that although we focused on registers, decisive linearizability is a general correctness criterion. Investigating its applicability beyond registers is left for future work. We believe that various implementations that are not strongly linearizable are still decisively linearizable (but, there are known implementations that are not even decisively linearizable, such as the Time-Stamped Stack [11], which allows concurrent complete push operations to remain unordered until a later pop determines their order). Identifying complete implementations for the class of decisively linearizable implementations of other objects is an important (and challenging!) avenue for future work. It would also be interesting to study (im)possibility for decisively linearizable implementations with different progress guarantees.

References

1 Yehuda Afek, Hagit Attiya, Danny Dolev, Eli Gafni, Michael Merritt, and Nir Shavit. Atomic snapshots of shared memory. *J. ACM*, 40(4):873–890, 1993. doi:10.1145/153724.153741.

2 Hagit Attiya, Amotz Bar-Noy, and Danny Dolev. Sharing memory robustly in message-passing systems. *J. ACM*, 42(1):124–142, 1995. doi:10.1145/200836.200869.

3 Hagit Attiya and Constantin Enea. Putting strong linearizability in context: Preserving hyperproperties in programs that use concurrent objects. In *DISC*, volume 146 of *LIPIcs*, pages 2:1–2:17, Dagstuhl, Germany, 2019. Schloss Dagstuhl – Leibniz-Zentrum für Informatik. URL: https://drops.dagstuhl.de/entities/document/10.4230/LIPIcs.DISC.2019.2, doi:10.4230/LIPIcs.DISC.2019.2.

4 Hagit Attiya and Constantin Enea. Putting strong linearizability in context: Preserving hyperproperties in programs that use concurrent objects. *CoRR*, abs/1905.12063, 2019. URL: `http://arxiv.org/abs/1905.12063`, `arXiv:1905.12063`.

5 Hagit Attiya, Constantin Enea, and Jennifer L. Welch. Impossibility of strongly-linearizable message-passing objects via simulation by single-writer registers. In *DISC*, volume 209 of *LIPIcs*, pages 7:1–7:18. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2021. URL: `https://doi.org/10.4230/LIPIcs.DISC.2021.7`, `doi:10.4230/LIPICS.DISC.2021.7`.

6 Hagit Attiya, Constantin Enea, and Jennifer L. Welch. Blunting an adversary against randomized concurrent programs with linearizable implementations. In *PODC*, pages 209–219. ACM, 2022. `doi:10.1145/3519270.3538446`.

7 Ahmed Bouajjani, Michael Emmi, Constantin Enea, and Suha Orhun Mutluergil. Proving linearizability using forward simulations. In *CAV*, volume 10427 of *LNCS*, pages 542–563. Springer, 2017. `doi:10.1007/978-3-319-63390-9\_28`.

8 David Yu Cheng Chan, Vassos Hadzilacos, Xing Hu, and Sam Toueg. An impossibility result on strong linearizability in message-passing systems. *CoRR*, abs/2108.01651, 2021. URL: `https://arxiv.org/abs/2108.01651`, `arXiv:2108.01651`.

9 Michael R. Clarkson and Fred B. Schneider. Hyperproperties. *J. Comput. Secur.*, 18(6):1157–1210, 2010. `doi:10.3233/JCS-2009-0393`.

10 John Derrick, Simon Doherty, Brijesh Dongol, Gerhard Schellhorn, and Heike Wehrheim. Brief announcement: On strong observational refinement and forward simulation. In *DISC*, volume 209 of *LIPIcs*, pages 55:1–55:4. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2021. URL: `https://doi.org/10.4230/LIPIcs.DISC.2021.55`, `doi:10.4230/LIPICS.DISC.2021.55`.

11 Mike Dodds, Andreas Haas, and Christoph M. Kirsch. A scalable, correct time-stamped stack. In *POPL*, pages 233–246. ACM, 2015. `doi:10.1145/2676726.2676963`.

12 Brijesh Dongol, Gerhard Schellhorn, and Heike Wehrheim. Weak progressive forward simulation is necessary and sufficient for strong observational refinement. In *CONCUR*, volume 243 of *LIPIcs*, pages 31:1–31:23. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2022. URL: `https://doi.org/10.4230/LIPIcs.CONCUR.2022.31`, `doi:10.4230/LIPICS.CONCUR.2022.31`.

13 Ivana Filipović, Peter O'Hearn, Noam Rinetzky, and Hongseok Yang. Abstraction for concurrent objects. *Theoretical Computer Science*, 411(51):4379–4398, 2010. URL: `https://www.sciencedirect.com/science/article/pii/S0304397510005001`.

14 Wojciech Golab, Lisa Higham, and Philipp Woelfel. Linearizable implementations do not suffice for randomized distributed computation. In *STOC*, pages 373–382, New York, NY, USA, 2011. ACM. `doi:10.1145/1993636.1993687`.

15 Vassos Hadzilacos, Xing Hu, and Sam Toueg. On register linearizability and termination. In *PODC*, pages 521–531. ACM, 2021. `doi:10.1145/3465084.3467925`.

16 Maryam Helmi, Lisa Higham, and Philipp Woelfel. Strongly linearizable implementations: possibilities and impossibilities. In *PODC*, pages 385–394. ACM, 2012. `doi:10.1145/2332432.2332508`.

17 Maurice P. Herlihy and Jeannette M. Wing. Linearizability: A correctness condition for concurrent objects. *ACM Trans. Program. Lang. Syst.*, 12(3):463–492, July 1990. `doi:10.1145/78969.78972`.

18 Prasad Jayanti, Siddhartha Jayanti, Ugur Yavuz, and Lizzie Hernandez. A universal, sound, and complete forward reasoning technique for machine-verified proofs of linearizability. *Proc. ACM Program. Lang.*, 8(POPL), jan 2024. `doi:10.1145/3632924`.

19 Nancy A. Lynch and Alexander A. Shvartsman. Robust emulation of shared memory using dynamic quorum-acknowledged broadcasts. In *FTCS*, pages 272–281. IEEE Computer Society, 1997. `doi:10.1109/FTCS.1997.614100`.

20 Gal Sela, Maurice Herlihy, and Erez Petrank. Brief announcement: Linearizability: A typo. In *PODC*, pages 561–564, New York, NY, USA, 2021. ACM. `doi:10.1145/3465084.3467944`.

21 Yoav Ben Shimon, Ori Lahav, and Sharon Shoham. Hyperproperty-preserving register specifications (extended version), 2024. URL: https://arxiv.org/abs/2408.11015, arXiv:2408.11015, doi:10.48550/arXiv.2408.11015.

A Proof Sketches

Proof (sketch) of Lemma 4.7. For downward closure, we observe that $I \sqsubseteq_{\mathsf{F}} I'$ implies that there exists a mapping $\pi : \mathsf{E}(I) \to \mathsf{E}(I')$ such that $\mathsf{h}(e) = \mathsf{h}(\pi(e))$ for every $e \in \mathsf{E}(I)$, and $\pi(e_1) \preceq_{\mathsf{P}} \pi(e_2)$ whenever $e_1 \preceq_{\mathsf{P}} e_2$. Then, given a suitable linearization mapping L for I', the composition $L \circ \pi$ is a suitable linearization mapping for I.

A complete implementation for $\mathcal{I}_R(O, \mathit{Spec})$ is similar to the complete implementation for the class of all linearizable implementations presented in the proof of Prop. 4.3, except that in addition to tracking in its internal state the history h it has observed so far, it also tracks a linearization s of h. When executing an invocation or response α, the linearization is non-deterministically updated to a linearization s' of $h \cdot \alpha$ such that $\langle s, s' \rangle \in R$. If such s' does not exist, the α step is not enabled. This construction generalizes the implementation in [4, Appendix C] which uses the same set of states but only allows to append actions to the linearization. ◀

Proof (sketch) of Theorem 5.4. To show that $\mathtt{WSR} \in \mathcal{I}_{\mathsf{ws}}$, we construct a linearization mapping $L : \mathsf{E}(\mathtt{WSR}) \to \mathsf{Spec}_{\mathsf{Reg}}$ by assigning "linearization points": Write operations that have already stored their values in X are linearized to the transition where they stored this value, and reads that picked a value to return are linearized to the transition where they first loaded that value. Other pending operations are not included in the linearization.

Hardness is much more challenging. Given a write strongly-linearizable implementation I, we begin by instrumenting the state of I with a ghost variable that tracks the full execution performed so far. Then, given an execution e and a transition t, we compare $L(e)$ and $L(e \cdot t)$, where L is the given write strong-linearization mapping for I. A naive attempt to show $I \sqsubseteq_{\mathsf{F}} \mathtt{WSR}$ would execute a store in $\mathtt{WSR}$ at the time the corresponding write operation w is added to the linearization. This fails since w might be added to linearization immediately after all previous writes have been linearized, which can be before w takes effect, and performing the store of $\mathtt{WSR}$ at that step will not allow later reads (that are concurrent with w) to load earlier values.

To overcome this, we prove the existence of a so-called *lazy* linearization mapping. Informally, this mapping adds operations to the linearization only when it must, e.g., when an operation completes, or when a write is needed to justify a completed read. More concretely, assuming arbitrary write strong-linearization mapping L, we prove the existence of a write strong-linearization mapping $L^* : \mathsf{E}(I) \to \mathsf{Spec}_{\mathsf{Reg}}$ with the following additional properties:

1. $L^*(e) = L^*(e \cdot t)$ for every $e \cdot t \in \mathsf{E}(I)$ such that the transition t is not labeled with a response.
2. For every $e \in \mathsf{E}(I)$ and operation o in $L^*(e)$, if o is not completed in $\mathsf{h}(e)$, then it is a write operation and it is not last in $L^*(e)$.
3. L^* is decisive.

Using L^*, the simulation works. Invocation and response transitions are simulated by an identical invocation or response, where invocations of read operations also load the stored value once. The stores in write operations are executed when the write operations are added to the lazy linearization, after which all pending reads that did not already load the stored value load it.

The crux of the proof is to justify that when a completed read is added to the linearization, the matching read in WSR has already loaded the value it needs to pick to match the return value of that read. For this, it suffices to show that the value the read returns was written by a write that either was rightmost in the linearizaton of the prefix of the execution up to the transition that invoked the read, or was added to the linearization later (as these are exactly the writes whose stores we load as described above). The properties of the lazy linearization are used to establish this fact. ◀

Proof (sketch) of Theorem 6.4. For inclusion, $\texttt{DR} \in \mathcal{I}_{\mathsf{d}}$, given an execution of DR we construct a linearization that only includes writes that already stored their value in X and reads that picked a value to return. This is similar to the linearization in the proof that $\texttt{WSR} \in \mathcal{I}_{\mathsf{ws}}$. However, the order in which the operations are linearized is more involved. We begin by assigning to each operation we intend to include in the linearization a version number: for a write operation it is the version number it wrote in the transition where it stored its value into X, and for a read operation it is the version number in the pre-state of the transition where it first loaded the value it later picked to return. Operations are ordered based on version number, with ties broken based on the ordering induced by the aforementioned transitions. The ordering between existing operations according to these rules does not change when new operations are added, and so the mapping we get is decisive. We can use the conditions guarding loads and "roll backs" to earlier version numbers to show the ordering according to the above rules respects real time order.

For hardness, the proof closely follows the proof that WSR is $\mathcal{I}_{\mathsf{ws}}$-hard. The main difference is that when a write appears for the first time in a linearization, it might not appear to the right of writes which already appeared earlier. We use the roll back mechanism to simulate these writes, thus maintaining an invariant that X contains the value of the rightmost write in the linearization. ◀

Proof (sketch) of Theorem 6.5. The simulation keeps track of when a pair of value v and timestamp ts reaches a majority of other processes for the first time. This can happen due to either a write distributing its newly written value or a read distributing its decided read value. When this happens, we check whether ts is larger than the current maximal timestamp that reached a majority of processes. If so, we perform in DR a store of v, attached to a new, larger version number, and then load this value with all threads that are active in a read method. Otherwise, we use the "overwritten value" path of DR: temporarily store v with a lower version number, collect this value by concurrent readers that can see it, and finally restore X to its latest value. ◀