Putting Strong Linearizability in Context: Preserving Hyperproperties in Programs That Use Concurrent Objects

Hagit Attiya
Technion – Israel Institute of Technology, Haifa, Israel
hagit@cs.technion.ac.il

Constantin Enea
Université de Paris, IRIF, CNRS, F-75013 Paris, France
cenea@irif.fr

Abstract

It has been observed that linearizability, the prevalent consistency condition for implementing concurrent objects, does not preserve some probability distributions. A stronger condition, called *strong linearizability* has been proposed, but its study has been somewhat ad-hoc. This paper investigates strong linearizability by casting it in the context of *observational refinement* of objects. We present a strengthening of observational refinement, which generalizes strong linearizability, obtaining several important implications.

When a concrete concurrent object *refines* another, more abstract object – often sequential – the correctness of a program employing the concrete object can be verified by considering its behaviors when using the more abstract object. This means that *trace properties* of a program using the concrete object can be proved by considering the program with the abstract object. This, however, does not hold for *hyperproperties*, including many security properties and probability distributions of events.

We define *strong observational refinement*, a strengthening of refinement that preserves hyperproperties, and prove that it is *equivalent* to the existence of *forward simulations*. We show that strong observational refinement generalizes *strong linearizability*. This implies that strong linearizability is also equivalent to forward simulation, and shows that strongly linearizable implementations can be composed both horizontally (i.e., *locality*) and vertically (i.e., with *instantiation*).

For situations where strongly linearizable implementations do not exist (or are less efficient), we argue that reasoning about hyperproperties of programs can be simplified by strong observational refinement of non-atomic abstract objects.

2012 ACM Subject Classification Theory of computation → Concurrency; Theory of computation → Program specifications; General and reference → Verification

Keywords and phrases Concurrent Objects, Linearizability, Hyperproperties, Forward Simulations

Digital Object Identifier 10.4230/LIPIcs.DISC.2019.2

Related Version Additional material can be found at `https://arxiv.org/abs/1905.12063`.

Funding *Hagit Attiya*: Partially supported by the Israel Science Foundation (1749/14 and 380/18). *Constantin Enea*: Supported in part by the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation program (grant agreement No 678177).

1 Introduction

Abstraction is key to the design and verification of large, complicated software. In concurrent programs, featuring intricate interactions between multiple threads, abstraction is often used to encapsulate low-level shared memory accesses within high-level abstract data types, called *concurrent objects*. Arguing about properties of such a program P is greatly simplified by

considering a concurrent object as a *refinement* of another, more abstract one: a *concrete* object O_1 is said to *observationally refine* another, *abstract* object O_2 if any behavior P can observe with O_1 is also observed by P with O_2. When O_2 is an *atomic object*, in which each operation is applied in exclusion, observational refinement is equivalent to linearizability [5, 12].[1]

Intuitively, linearizability, and more generally, observational refinement, seem to imply that anything we can prove about P with O_2 also holds when P executes with O_1. This is indeed the case when considering *trace properties*, i.e., properties that are specified as *sets of traces*, in particular, safety properties.

Unfortunately, many interesting properties cannot be specified as properties of individual traces, i.e., as trace properties. Notable examples are security properties such as *noninterference* [13], stipulating that commands executed by users with high clearance have no effect on system behavior observed by users with low clearance. Other examples are quantitative properties like bounds on the *probability distribution* of events, e.g., the mean response time over sets of executions. Indeed, while the fact that *the average response time of an operation in an execution is smaller than some bound X* is a trace property, the requirement that the *average response time over all executions is smaller than X* cannot be stated as a trace property.

Hyperproperties [9], namely, sets of sets of traces, allow to capture such expectations. By definition, every property of system behavior (for systems modeled as trace sets) can be specified as a hyperproperty. It is known that observational refinement does not preserve hyperproperties [18], in general. More recently, it has been shown that linearizability does not preserve probability distributions over traces [14], allowing an adversary scheduler additional control over the possible outcomes of a distributed randomized program. (An example appears in Section 2.)

This paper defines the notion of *strong observational refinement*, relates it to hyperproperties, and shows its equivalence to *forward simulations*. We show that strong observational refinement generalizes strong linearizability [14].[2] We also explore the possibility of using – instead of the classical sequential specifications – *concurrent* specifications, which are nevertheless simpler.

To explain our results in more detail, consider a *labeled transition system* (*LTS*) that, intuitively, represents all the executions of the object under the most general client (that may call methods in any order and from any thread). A state of the LTS corresponds to a state of the object and transitions correspond to method calls/returns, or internal steps within a method invocation. A *sequential specification* corresponds to a concurrent object where essentially, method bodies consist of a single atomic step that acts according to the sequential specification (hence, they are totally ordered in time during any execution).

An LTS O_1 *observationally refines* an LTS O_2 if and only if the *histories* (i.e., sequences of call/return actions) generated by O_1 are included in those generated by O_2 [5]. In this way, observational refinement of two LTSs reduces to a inclusion between their traces, when projected over some alphabet Γ (in this case, Γ is the set of call/return actions), called Γ-*refinement*.

A *forward simulation* maps every step of O_1 to a sequence of steps of O_2, starting from the initial state of O_1 and advancing in a forward manner; a *backward simulation* is similar, but it goes in the reverse direction, from end states back to initial states. When proving

[1] *Linearizability* [16] states that a concurrent execution of operations corresponds to some serial sequence of the same operations permitted by the specification.

[2] *Strong linearizability* requires that *the linearization of a prefix of a concurrent execution is a prefix of the linearization of the whole execution*, see Section 5.

```
a = push(0);          ||  b = push(1);          ||  assume a == b == OK;
low1 = pop();             low2 = pop();             push(2);
                                                    high = highBooleanInput();
```

Figure 1 A program with three threads using a concurrent stack (we assume that `push` returns the value `OK`). Statements in the same thread are aligned vertically. The statement `assume` blocks the program when the Boolean condition is not satisfied and `highBooleanInput` returns a Boolean value labeled as high clearance. The `assume` statement enforces that `push(0)` and `push(1)` finish before `push(2)` starts.

linearizability, an important special case of forward simulation is the identification of *fixed* linearization points. A forward/backward simulation can be parameterized by an alphabet Γ, in which case the sequence of steps of O_2 associated to a step of O_1 should contain a step labeled by an action $a \in \Gamma$ if and only if the step of O_1 is also labeled by a. It is known [17] that Γ-refinement holds if and only if there is a combination of Γ-forward and Γ-backward simulations from O_1 to O_2; a forward simulation suffices when the projection of O_2 on Γ is deterministic. (See Section 3.)

The notion of strong observational refinement relies on the concept of a *deterministic scheduler* that resolves the non-determinism introduced for instance, from the execution of internal actions by parallel threads (it is similar to the notion of strong adversary introduced in the context of randomized algorithms [4]). O_1 *strongly (observationally) refines* O_2 if a program P running under a deterministic schedule with O_1 makes the same observations as when P runs with O_2 with a possibly-different deterministic schedule. (The complete definition appears in Section 4.) We prove that strong observational refinement implies the existence of a forward simulation. The converse direction is fairly straightforward, proving the equivalence of these two notions, and imply compositional proof methodologies. (These results appear in Section 5.) By relating strong linearizability to strong observational refinement, we prove that a concrete object is a strong linearization of an atomic object *if and only if* there is an appropriate forward simulation between the two. This immediately implies methods for composing strongly linearizable concurrent objects.

To address situations where there is no concrete object that strongly linearizes a particular atomic object [15, 10], or in cases where such an object is less efficient, we suggest concrete objects that strongly refine other, more abstract objects that still expose some concurrency. This follows [6] and allows to simplify reasoning about randomized programs even when using objects like the Herlihy&Wing queue [16] or snapshot objects [1], which are not strongly linearizable. For example, in the case of atomic snapshots, the abstract object that obtains several instantaneous snapshots during the *scan* operation and then *arbitrarily* returns one of them (see Section 6). Arguing about a program using this abstract object is simpler, while still exposing the power of an adversarial scheduler to manipulate the responses of a scan.

2 Motivating Example: A Stack Implementation that Leaks Information

When an object O_1 refines a specification object O_2, any safety property of a program P using O_2 (that refers only to P's state and is agnostic to the internal state of the object O_2) is preserved when O_2 is replaced by its refinement O_1. However, refinement does not preserve *hyperproperties* [9], i.e., properties of *sets* of traces.

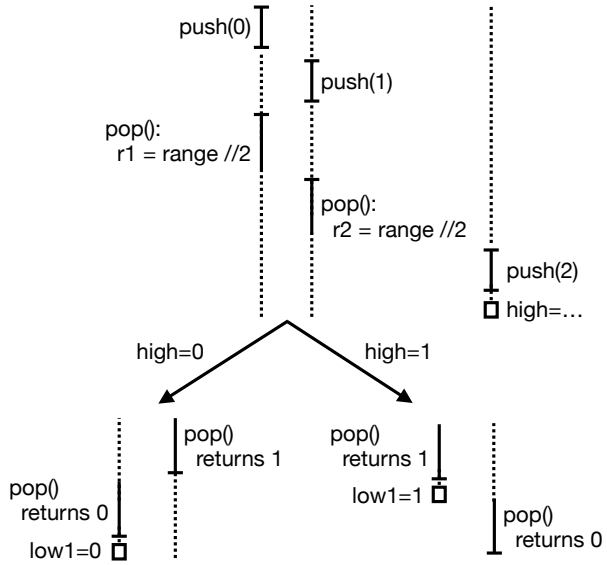

Figure 2 A scheduler for the program in Figure 1. Time flows from top to bottom. Dotted edges denote periods of time where a thread is not active.

We explain this issue by considering *noninterference* [13] in the program of Figure 1. This program invokes methods of a concurrent stack, and we wish to show that independently of the thread scheduler, none of the low clearance variables `low1` and `low2` can leak the value of the high clearance variable `high`, i.e., it is impossible to define a thread scheduler which admits only executions where `low1` = `high` or only executions where `low2` = `high`. A precise notion of scheduler will be defined below, but for now, it is enough to think of a thread scheduler as a monitor that chooses to activate threads depending on the history of the execution. This property is satisfied by the program when invoking an *atomic* concurrent stack. Indeed, assuming that `push(0)` is scheduled before the one of `push(1)` (the other case is similar), then either (i) `push(2)` is scheduled before at least one of the `pop` invocations, and then, $\texttt{low1}, \texttt{low2} \in \{1, 2\}$ which shows that none of these two variables equals the value of `high` when `high` = 0, or (ii) `push(2)` is scheduled after the `pop` invocations, and then, the scheduler admits executions where `low1` = b_1, `low2` = b_2, and `high` = 0 if and only if it admits executions where `low1` = b_1, `low2` = b_2, and `high` = 1 (for $b_1, b_2 \in \{0, 1, 2\}$).

This property is however not satisfied by this program when using the concurrent stack of Afek et al. [2]. This stack stores the elements into an infinite array `items`; a shared variable `range` keeps the index of the first unused position in `items`. The push method stores the input value in the array while also incrementing `range` (the details are irrelevant for our example). The pop method first reads `range` and then traverses the array backwards starting from the predecessor of this position, until it finds a position storing a non-null element (array cells can be nullified by concurrent pop invocations). It atomically reads this element and stores null in its place. If the pop reaches the bottom of the array without finding non-null cells, then it returns that the stack is empty. Unlike the case of atomic stacks, Figure 2 shows a thread scheduler where `low1` stores the value of `high`. This scheduler imposes that `push(0)` executes before `push(1)` (so that 0 occurs before 1 in the array `items`),[3] and then

[3] A similar scheduler can be defined when `push(1)` executes before `push(0)`.

preempts pop invocations just after reading the value of `range` which equals 2 (assuming the array indexing starts at 0). Then, it schedules the third thread and, depending on the value of `high`, it schedules the rest of the `pop` invocations such that the `pop` in the first thread extracts a value which equals `high`. This ensures that `low1` == `high`.

This shows that noninterference in programs invoking the atomic stack is not preserved when the latter is replaced by the concurrent stack of Afek et al. [2], although the latter is a refinement of the atomic stack. Section 4 presents a stronger notion of observational refinement that preserves hyperproperties and in particular, noninterference.

3 Modelling Concurrent Objects as Labeled Transition Systems

Labeled transition systems (*LTS*) capture shared-memory programs with an arbitrary number of threads, abstracting away the details of any particular programming system irrelevant to our development.

An LTS $A = (Q, \Sigma, s_0, \delta)$ over the possibly-infinite alphabet Σ is a possibly-infinite set Q of states with initial state $s_0 \in Q$, and a transition relation $\delta \subseteq Q \times \Sigma \times Q$. The ith symbol of a sequence $\tau \in \Sigma^*$ is denoted τ_i, and ϵ is the empty sequence. An *execution* of A is an alternating sequence of states and transition labels (also called *actions*) $\rho = s_0, a_0, s_1 \ldots a_{k-1}, s_k$ for some $k > 0$ such that $(s_i, a_i, s_{i+1}) \in \delta$ for each $0 \leq i < k$. We write $s_i \xrightarrow{a_i \ldots a_{j-1}}_A s_j$ as shorthand for the subsequence $s_i, a_i, ..., s_{j-1}, a_{j-1}, s_j$ of ρ. (in particular $s_i \xrightarrow{\epsilon} s_i$).

The projection $\tau|\Gamma$ of a sequence τ is the maximum subsequence of τ over alphabet Γ. This notation is extended to sets of sequences as usual. A *trace* of A is the projection $\rho|\Sigma$ of an execution ρ of A. The set of executions of an LTS A is denoted by $E(A)$, while the set of traces of A is denoted $T(A)$. An LTS is *deterministic* if for any state s and any sequence $\tau \in \Sigma^*$, there is at most one state s' such that $s \xrightarrow{\tau} s'$. More generally, for an alphabet $\Gamma \subseteq \Sigma$, an LTS is Γ-*deterministic* if for any state s and any sequence $\tau \in \Gamma^*$, there is at most one state s' such that $s \xrightarrow{\tau'} s'$ and τ is a subsequence of τ'.

An *object* is a *deterministic* LTS over alphabet $C \cup R \cup \Sigma_o$ where C is the set of call actions, R is the set of return actions, and Σ_o is an alphabet of internal actions. Formally, a call action $call(m, d, k)$ combines a method m and argument d with an operation identifier k, while a return action $ret(m, d, k)$ combines a method m and return value d with an operation identifier k. Operation identifiers are used to pair call and return actions. We assume that the traces of an object satisfy standard well-formedness properties, e.g., return actions correspond to previous call actions. Given a standard description of an object implementation as a set of methods, its LTS represents the executions of its most general client (that may call methods in any order and from any thread). The states of the LTS represent the shared state of the object together with the local state of each thread. The transitions correspond to statements in the method bodies (in which case they are labeled by internal actions in Σ_o), or call and return actions. For simplicity, we ignore the association of method invocations to threads since it is irrelevant to our development. A trace τ of an object O projected over call and return actions is called a *history* of O, and it is denoted by $hist(\tau)$. The set of histories admitted by an object O is denoted by $H(O)$. Call and return actions $call(m, \_, k)$ and $ret(m, \_, k)$ are called *matching* when they contain the same operation identifier. A call action is called *unmatched* in a history h when h does not contain the matching return. A history h is called *sequential* if every call $call(m, d, k)$ is immediately followed by the matching return $ret(m, \_, k)$. Otherwise, it is called *concurrent*.

Linearizability [16] is a standard correctness criterion for concurrent objects expressing conformance to a given sequential specification. This criterion is based on a relation $\sqsubseteq$ between histories: $h_1 \sqsubseteq h_2$ iff there exists a well-formed history h_1' obtained from h_1 by appending

return actions that correspond to unmatched call actions in h_1 or deleting unmatched call actions, such that h_2 is a permutation of h_1' that preserves the order between return and call actions, i.e., a given return action occurs before a given call action in h_1' iff the same holds in h_2. We say that h_2 is a *linearization* of h_1. A history h_1 is called *linearizable* w.r.t. an object O_2 iff there exists a sequential history $h_2 \in H(O_2)$ such that $h_1 \sqsubseteq h_2$. An object O_1 is linearizable w.r.t. O_2, written $O_1 \sqsubseteq O_2$, iff each history $h_1 \in H(O_1)$ is linearizable w.r.t. O_2.

Linearizability has been shown equivalent to a criterion called *observational refinement* which states that every behavior of every program possible using a concrete object would also be possible were the abstract object used instead [5, 12] (the precise meaning of behavior is given below). Actually, this result holds only when the abstract object is *atomic*, i.e., an implementation where the methods of a sequential object are guarded by a global-lock acquisition. Formally, given a set of *sequential* histories Seq, an *atomic* object is an LTS $O = (Q, \Sigma, s_0, \delta)$ where the states are pairs formed of a history h and a linearization h_s of h, i.e., $Q = \{(h, h_s) : h_s \in Seq \text{ and } h \sqsubseteq h_s\}$, the internal actions are *linearization point* actions $lin(k)$ (for linearizing an operation with identifier k), i.e., $\Sigma = C \cup R \cup \{lin(k) : k \in \mathbb{I}\}$ where $\mathbb{I}$ denotes the set of operation identifiers, the initial state contains an empty history and linearization, $s_0 = (\epsilon, \epsilon)$, and the transition relation is defined by: $((h, h_s), a, (h', h_s')) \in \delta$ if

$$a \in C \implies h' = h \cdot a \text{ and } h_s' = h_s$$
$$a \in R \implies h' = h \cdot a \text{ and } h_s' = h_s \text{ and } a \text{ occurs in } h_s'$$
$$a = lin(k) \implies h' = h \text{ and } h_s' = h_s \cdot call(m, d_1, k) \cdot ret(m, d_2, k), \text{ for some } m,\ d_1, \text{ and } d_2.$$

Call actions are only appended to the history h, return actions ensure that additionally, the linearization h_s' contains the corresponding operation, and linearization points extend the linearization with a new operation. Note that O admits every history which is linearizable w.r.t. Seq, i.e., $H(O) = \{h : \exists h' \in Seq.\ h \sqsubseteq h'\}$.

A *program* is a *deterministic* LTS over alphabet $C \cup R \cup \Sigma_p$ where Σ_p is an alphabet of *program actions*. Program actions can be interpreted for instance, as assignments to some program variables which are disjoint from the variables used by the object, or as different outcomes of a random choice. The executions of a program P with an object O are obtained as the executions of the LTS product $P \times O$ [4]. As program and object alphabets only intersect on call and return actions, our formalization supposes that programs and objects communicate only through method calls and returns, and not, e.g., through additional shared random-access memory.

Observational refinement between objects O_1 and O_2 means that any "observation" extracted from a program execution possible with O_1 (referred to as a "concrete" object), is also possible with O_2 (referred to as the "specification"). We define observations as projections of traces over program actions. The methods invoked by a program along with their inputs and return values can be recorded using additional program actions. In the context of a concrete programming language, one can use additional program variables to record the inputs before calling a method and the return values upon their return.

▶ **Definition 1.** *The object* O_1 observationally refines O_2*, written* $O_1 \preceq O_2$*, iff*

$$T(P \times O_1)|\Sigma_p \subseteq T(P \times O_2)|\Sigma_p$$

for all programs P *over alphabet* $\Sigma_p \cup C \cup R$*.*

[4] The *product* $A_1 \times A_2$ of two LTSs is defined as usual, respecting $E(A_1 \times A_2)|(\Sigma_1 \cap \Sigma_2) = E(A_1)|\Sigma_2 \cap E(A_2)|\Sigma_1$.

The following theorem relates observational refinement to a standard notion of refinement between LTSs, defined roughly as inclusion of traces, in the context of concurrent objects.[5] For two LTSs A_1 and A_2, we say that A_1 *refines* A_2 when $T(A_1) \subseteq T(A_2)$. More generally, for an alphabet Γ, A_1 Γ-*refines* A_2 when $T(A_1)|\Gamma \subseteq T(A_2)|\Gamma$. By an abuse of notation, $A_1 \subseteq_\Gamma A_2$ denotes the fact that A_1 Γ-refines A_2 (we will omit Γ when it is understood from the context). Intuitively, the alphabet Γ represents a set of actions which are "observable" in both A_1 and A_2, the actions not in Γ are considered to be "internal" to A_1 or A_2. Observational refinement is equivalent to $(C \cup R)$-refinement which means that the histories of the concrete object are included in those of the specification (note that "plain" refinement would not hold because the internal actions may differ).

▶ **Theorem 2** ([5, 12]). *$O_1 \leq O_2$ iff $O_1 \subseteq_{C \cup R} O_2$. If O_2 is atomic, then $O_1 \leq O_2$ iff $O_1 \sqsubseteq O_2$.*

In the rest of the paper, since observational refinement and $(C \cup R)$-refinement are equivalent, we will not make the distinction between the two and refer to both as *refinement*.

4 Strong Observational Refinement

As discussed in Section 2, refinement does not preserve hyperproperties, which are properties of *sets* of traces and not individual traces as in the case of safety properties. In the following, we define a stronger notion of observational refinement that preserves such properties, using a notion of scheduler that is actually just a mechanism for resolving the non-determinism induced by internal actions, irrespectively of whether it comes from executing a set of parallel threads.

A *scheduler* for a deterministic LTS $A = (Q, \Sigma, s_0, \delta)$ over alphabet Σ is a function $S : \Sigma^* \to 2^\Sigma$ which prescribes a possible set of next actions to continue an execution based on a sequence of previous actions. A trace $\tau = a_0 \cdot \ldots \cdot a_{k-1}$ is *consistent* with a scheduler S if $a_i \in S(a_0 \cdot \ldots \cdot a_{i-1})$ for all $0 \leq i < k$ (where by an abuse of notation, $a_0 \cdot a_{-1}$ represents the empty sequence). An execution is consistent with a scheduler S if its trace is. The set of executions of an LTS A consistent with a scheduler S can be defined using an LTS which is the product between A and an LTS A_S whose states are sequences in Σ^* and the transitions link a state $\tau \in \Sigma^*$ to a state $\tau \cdot a \in \Sigma^*$ provided that $a \in S(\tau)$ (such a transition is labeled by a). Let $T(A, S)$ denote the set of traces of A consistent with S. A scheduler is *admitted* by A if for every k, if $\tau = a_0 \cdot \ldots \cdot a_{k-1}$ is a trace of A consistent with S, then $S(a_0 \cdot \ldots \cdot a_{k-1})$ is non-empty and every $a \in S(a_0 \cdot \ldots \cdot a_{k-1})$ is enabled in the state s_k with $s_0 \xrightarrow{a_0 \cdot \ldots \cdot a_{k-1}}_A s_k$.

A scheduler of an LTS $P \times O$ (the product of a program P and an object O) is called *deterministic* when it fixes in a unique way the actions of O to continue an execution, i.e., for every sequence τ, $S(\tau) \subseteq \Sigma_p$ or $|S(\tau)| = 1$ (where Σ_p is the set of program actions). When program actions represent outcomes of random choices made by the program, a deterministic scheduler can be used to model a strong adversary [4] which schedules threads depending on those outcomes. An object O_1 strongly (observationally) refines an object O_2 if any deterministic schedule admitted by a program P when using O_1 leads to exactly the same set of "observations" as a deterministic schedule admitted by P were O_2 used instead. Formally,

[5] This relationship has been shown under natural assumptions about objects and programs [5]. For instance, concerning objects, it is assumed that call actions cannot be disabled and they cannot disable other actions (they can be reordered to the left while preserving the computation), and return actions cannot enable other actions.

▶ **Definition 3.** *The object* O_1 strongly (observationally) refines O_2*, written* $O_1 \preceq_s O_2$*, iff*

for every deterministic scheduler S_1 *admitted by* $P \times O_1$*,*
there exists a deterministic scheduler S_2 *admitted by* $P \times O_2$*,*
such that $T(P \times O_1, S_1)|\Sigma_p = T(P \times O_2, S_2)|\Sigma_p$
for all programs P *over alphabet* $\Sigma_p \cup C \cup R$*.*

A *hyperproperty* of a program P over alphabet $\Sigma_p \cup C \cup R$ is a set of sets of sequences over Σ_p. For instance, the hyperproperty discussed in the context of the program in Figure 1 is the set of all sets T s.t.

$(\exists \tau \in T.\ \texttt{low1}(\tau) \neq \texttt{high}(\tau)) \wedge (\exists \tau \in T.\ \texttt{low2}(\tau) \neq \texttt{high}(\tau))$

where for any variable x, $x(t)$ is the value of x at the end of trace t. A hyperproperty φ is *satisfied* by a program P with an object O, written $P \times O \vDash \varphi$, if $T(P \times O, S)|\Sigma_p \in \varphi$ for every deterministic scheduler S.

▶ **Theorem 4.** *If* $O_1 \preceq_s O_2$*, then* $P \times O_2 \vDash \varphi$ *implies* $P \times O_1 \vDash \varphi$ *for every hyperproperty* φ *of* P*.*

Proof. Assume that $O_1 \preceq_s O_2$ and $P \times O_2 \vDash \varphi$ for some hyperproperty φ of P. Let S_1 be a deterministic scheduler admitted by $P \times O_1$. Since $O_1 \preceq_s O_2$, there exists a deterministic scheduler S_2 admitted by $P \times O_2$ such that $T(P \times O_1, S_1)|\Sigma_p = T(P \times O_2, S_2)|\Sigma_p$. Since, $P \times O_2 \vDash \varphi$, we get that $T(P \times O_2, S_2)|\Sigma_p \in \varphi$, which implies that $T(P \times O_1, S_1)|\Sigma_p \in \varphi$. Therefore, $P \times O_1 \vDash \varphi$. ◀

This preservation result applies to *probabilistic* hyperproperties as well, for instance when reasoning about randomized consensus protocols [4]. Since a deterministic scheduler fixes in a unique way the object's actions to continue an execution, probability distributions can be assigned only to actions which are internal to the program P. This holds for randomized protocols, where randomization is due to coin flip operations that are internal to the protocol and do not concern the behavior of the objects it invokes. Then, the probabilities associated with program actions can be encoded in the action names, thereby encoding probabilistic (hyper)properties as properties of (sets of) traces (see [9] for more details).

5 Characterizing Strong Refinement Using Forward Simulations

In general, proving refinement between two LTSs relies on *simulation relations* which roughly, are relations between the states of the two LTSs showing that one can mimic every step of the other one. *Forward* simulations show that every outgoing transition from a given state can be mimicked by the other LTS while *backward* simulations show the same for every incoming transition to a given state. Applying induction, forward simulations show that every trace of an LTS is admitted by the other LTS starting from initial states and advancing in a forward manner, while backward simulations consider the backward direction, from end states to initial states. It has been shown that (Γ-)refinement is equivalent to the existence of a composition of forward and backward simulations, and to the existence of only a forward simulation provided that the specification[6] is (Γ-)deterministic [17]. In the following, we show that strong observational refinement is equivalent to the existence of a *forward* simulation, which implies that refinement is strictly weaker than strong observational refinement (forward simulations do not suffice to establish refinement in general).

[6] When an LTS A_1 (Γ-)refines another LTS A_2, we refer to A_2 as the specification.

▶ **Definition 5.** *Let* $A_1 = (Q_1, \Sigma_1, s_0^1, \delta_1)$ *and* $A_2 = (Q_2, \Sigma_2, s_0^2, \delta_2)$ *be two LTSs and* Γ *an alphabet. A relation* $F \subseteq Q_1 \times Q_2$ *is called a* Γ-forward simulation *from* A_1 *to* A_2 *iff* $(s_0^1, s_0^2) \in F$ *and:*

- *for all* $s_1, s_1' \in Q_1$, $a \in \Sigma_1$, *and* $s_2 \in Q_2$, *such that* $(s_1, a, s_1') \in \delta_1$ *and* $(s_1, s_2) \in F$, *we have that there exists* $s_2' \in Q_2$ *and* $\tau \in \Sigma_2^*$ *such that* $(s_1', s_2') \in F$ *and* $s_2 \xrightarrow{\tau}_{A_2} s_2'$ *and* $\tau|\Gamma = a|\Gamma$.

A Γ-forward simulation states that every step of A_1 is simulated by a sequence of steps of A_2 (this sequence can be empty to allow for stuttering). Since it should imply that A_1 Γ-refines A_2, every step of A_1 labeled by an observable action $a \in \Gamma$ should be simulated by a sequence of steps of A_2 where exactly one transition is labeled by a and all the other transitions are labeled by non-observable actions (this is implied by $\tau|\Gamma = a|\Gamma$). Also, every internal step of A_1 should be simulated by a sequence of internal steps of A_2.

An instantiation of forward simulations are linearizability proofs using the so-called "fixed linearization points". Linearizability of a history can be proved by showing that each invocation can be seen as happening at some point, called linearization point, occurring somewhere between the call and return actions of that invocation. Then, the linearization points are *fixed* when they are mapped to a certain fixed set of statements (usually, one statement per method). This defines a mapping between steps of a concrete implementation and steps of an atomic object, i.e., those fixed statements map to linearization point actions in the atomic object and all the other statements correspond to stuttering steps of the atomic object, thereby defining a forward simulation between the two. As a side remark, backward simulation is necessary to prove linearizability w.r.t. atomic specifications, when linearization points depend on future steps in the execution, the Herlihy&Wing queue [16] being a classic example (Schellhorn et al. [21] present such a proof).

The easier direction is showing that forward simulations imply strong refinement. A forward simulation from O_1 to O_2 can be used to simulate any scheduler S_1 of a program P using O_1 by a scheduler of the same program P when using O_2. Program actions will be replayed exactly as in S_1 while the actions of O_2 simulating actions of O_1 can be chosen according to the forward simulation.

▶ **Lemma 6.** *If there exists a* $(C \cup R)$*-forward simulation from* O_1 *to* O_2*, then* $O_1 \leq_s O_2$.

Proof. Let $O_1 = (Q_1, \Sigma_1, s_0^1, \delta_1)$ and $O_2 = (Q_2, \Sigma_2, s_0^2, \delta_2)$ be two objects, and F a $(C \cup R)$-forward simulation from O_1 to O_2. Let P be a program over alphabet $\Sigma_p \cup C \cup R$ and S_1 a deterministic scheduler admitted by $P \times O_1$. We define a rewriting relation $\rightsquigarrow$ between traces of $P \times O_1$ consistent with S_1 and traces of $P \times O_2$ such that intuitively, if $\tau \rightsquigarrow \tau'$ then τ' is a trace of $P \times O_2$ which simulates the trace π of $P \times O_1$ with respect to the simulation relation F. Formally, $\rightsquigarrow$ is the smallest relation satisfying the following:

- $\epsilon \rightsquigarrow \epsilon$
- if $\tau \rightsquigarrow \tau'$, $S_1(\tau) \subseteq \Sigma_p$, and $a \in S_1(\tau)$, then $\tau \cdot a \rightsquigarrow \tau' \cdot a$,
- if $\tau \rightsquigarrow \tau'$, $S_1(\tau) \cap \Sigma_p = \varnothing$, and $a = S_1(\tau)$ (in this case, $a \in \Sigma_1$ and $S_1(\tau)$ is a singleton because S_1 is deterministic), then $\tau \cdot a \rightsquigarrow \tau' \cdot F(S_1(\tau))$, where $F(S_1(\tau))$ is a sequence of actions of O_2 simulating the action $a = S_1(\tau)$ in the state reached after the trace $\tau|\Sigma_1$. Formally, if $s_0^1 \xrightarrow{\tau|\Sigma_1}_{O_1} s_1$, then a simple induction on the length of executions can show that $(s_1, s_2) \in F$ where $s_0^2 \xrightarrow{\tau'|\Sigma_2}_{O_2} s_2$. Then, since $s_1 \xrightarrow{S_1(\tau)}_{O_1} s_1'$ is a transition of O_1 and F is a forward simulation, we get that there exists s_2' such that $(s_1', s_2') \in F$ and $s_2 \xrightarrow{\sigma}_{O_2} s_2'$ and $\sigma|(C \cup R) = S_1(\tau)|(C \cup R)$. We define $F(S_1(\tau)) = \sigma$.

Then, we define a deterministic scheduler S_2 admitted by $P \times O_2$ inductively as follows:

$$S_2(\epsilon) = S_1(\epsilon) \qquad \text{if } \tau \rightsquigarrow \tau', \text{ then } S_2(\tau') = \begin{cases} S_1(\tau), \text{if } S_1(\tau) \in \Sigma_p \\ F(S_1(\tau)), \text{otherwise} \end{cases}$$

Note that S_2 is a slight deviation from the definition of a scheduler because $F(S_1(\tau))$ is not necessarily a single action, but a sequence of actions. However, the definition of S_2 can be adapted easily such that this sequence of steps is performed one by one. For any other sequence τ' which is not considered in the definition above, $S_2(\tau')$ is defined arbitrarily.

Since F is a $(C \cup R)$-forward simulation, $T(P \times O_1, S_1)|\Sigma_p \subseteq T(P \times O_2, S_2)|\Sigma_p$ is obvious. The reverse, i.e., $T(P \times O_2, S_2)|\Sigma_p \subseteq T(P \times O_1, S_1)|\Sigma_p$, follows from the fact that S_2 is defined inductively following the definition of S_1. ◀

We now prove our key technical result: strong observational refinement (from O_1 to O_2) implies the existence of a $(C \cup R)$-forward simulation (from O_1 to O_2). Since the latter implies refinement, a corollary of this result is that strong observational refinement implies observational refinement. Thus, we define a program P which corresponds to the most general client (of O_1) and which uses particular program actions to guess the possible continuations of a given execution with call and return actions. Then, we define a scheduler S_1 which ensures that the executions of P with O_1 are consistent with the guesses made by the program. By strong observational refinement, there exists a scheduler S_2 such that P produces the same sequences of "guess" actions and call/return actions when using O_2 and constrained by S_2 as when using O_1 and constrained by S_1 (since strong observational refinement considers traces projected over program actions, the preservation of call/return actions is not guaranteed explicitly, but it can be enforced using additional program actions used to record them). If Γ is the union of the set of "guess" actions and the set of call/return actions, then the program P used in conjunction with O_2 and constrained by the scheduler S_2 is Γ-deterministic. Therefore, there exists a forward simulation between the two variations of P. Because the program states are disjoint from the object states, this forward simulation between programs leads to a forward simulation between objects.

▶ **Lemma 7.** *If $O_1 \leq_s O_2$, then there exists a $(C \cup R)$-forward simulation from O_1 to O_2.*

Proof. Let $O_1 = (Q_1, \Sigma_1, s_0^1, \delta_1)$ and $O_2 = (Q_2, \Sigma_2, s_0^2, \delta_2)$ be two objects. Also, let $\Sigma_p = \{record(a), guess(H) : a \in C \cup R, H \subseteq (C \cup R)^*\}$ be a set of program actions for recording a call/return action a ($record(a)$) or guessing a set H of possible continuations with sequences of call/return actions ($guess(H)$). We define a program P with a single state and self-loop transitions labeled by all symbols in $\Sigma_p \cup C \cup R$, i.e., $P = (\{s_0\}, \Sigma_p \cup C \cup R, s_0, \delta)$ where $(s_0, \alpha, s_0) \in \delta$ for all $\alpha \in \Sigma_p \cup C \cup R$.

We define a deterministic scheduler S_1 which ensures that the guesses made by P when using O_1 are correct, and that the call/return actions are tracked correctly using *record* actions. To ensure the correctness of guesses, we define a mapping $after_1 : Q_1 \to 2^{(C \cup R)^*}$ which associates every state s with the set of call/return sequences admitted from s, i.e., $after_1(s) = \{\sigma : \sigma \in (C \cup R)^*, \exists \tau, s'.\ s \xrightarrow{\tau}_{O_1} s' \wedge \tau|(C \cup R) = \sigma\}$.

Let S_1 be a deterministic scheduler such that for every $a_0, \ldots, a_{k-1} \in \Sigma_1$ and $k \geq 0$,

$$S_1(a_0 \cdot \ldots \cdot a_{k-1}) = \{record(a_{k-1})\} \text{ if } a_{k-1} \in C \cup R \text{ and } k \geq 1$$

$$S_1(a_0 \cdot \ldots \cdot a_{k-1}[\cdot record(a_{k-1})]) = \{guess(H) : \exists a.\ s_0^1 \xrightarrow{a_0 \cdot \ldots \cdot a_{k-1}|\Sigma_1}_{O_1} s \xrightarrow{a}_{O_1} s' \text{ and } H = after_1(s')\} \text{ if } a_{k-1} \notin \Sigma_p$$

$$S_1(a_0 \cdot \ldots \cdot a_{k-1} \cdot guess(H)) = \{a\}, \text{ for some } a \in \Sigma_1 \text{ s.t. } s_0^1 \xrightarrow{a_0 \cdot \ldots \cdot a_{k-1} \cdot a|\Sigma_1}_{O_1} s \text{ and } H = after_1(s)$$

Informally, the first rule enforces that every call/return action a is followed by a program action $record(a)$. The second rule ensures that S_1 is permissive enough, i.e., it allows all the successors of the current object state that have different $after_1$ images. More precisely, for every sequence σ ending in an internal object action $a_{k-1} \in \Sigma_1 \setminus (C \cup R)$ or the sequence $a_{k-1} \cdot record(a_{k-1})$ when $a_{k-1} \in C \cup R$ (we use $\sigma[\cdot a]$ to denote a sequence where the character a is optional), S_1 schedules every $guess(H)$ action where H is the $after_1$ image of a successor of the current object state. The third rule ensures that every $guess(H)$ is followed by an action leading to an object state s with $H = after_1(s)$. The last two cases ensure that every action a of O_1 is preceded by a $guess(H)$ program action where H is the set of call/return sequences admitted from the post-state of a.

Although S_1 does not admit all the executions of O_1 (because of the arbitrary choice of a in the third case above), we show that the set of executions it admits simulate all the executions of O_1: let $O_1[S_1]$ be an LTS representing the set of executions of O_1 consistent with S_1 (obtained from the set of executions of P consistent with S_1 by projecting out the program state and actions). We show that the relation F_1 between states of O_1 and $O_1[S_1]$, respectively, defined by $(s, s') \in F_1$ iff $after_1(s) = after_1(s')$, is a $(C \cup R)$-forward simulation from O_1 to $O_1[S_1]$. The fact that it relates the initial object states s_0^1 and s_0^1 is trivial. Now, let $s, s_1 \in Q_1$ and $a \in \Sigma_1$ such that $(s, a, s_1) \in \delta_1$ and $(s, s') \in F_1$. Using a simple induction on the length of executions, it can be shown that there exists a state s_1' with $after_1(s_1) = after_1(s_1')$ such that (s', b, s_1') for some action b. If $a \in C \cup R$, then $b = a$ because otherwise, the continuations with call/return actions admitted from s_1 will be different from those admitted from s_1' (for instance, if a is a call action and b is an internal action, then the matching return action will be eventually enabled in executions starting from s_1 but not from s_1', at least not before a occurs). For the same reason, if a is an internal action, then b is also an internal action. This concludes the proof that F_1 is a forward simulation.

Since $O_1 \leq_s O_2$, there exists a scheduler S_2 such that $T(P \times O_1, S_1)|\Sigma_p = T(P \times O_2, S_2)|\Sigma_p$. Let $P[O_2, S_2]$ denote the LTS representation of the set of executions of P with O_2 and consistent with S_2 (explained in Section 4). It can be easily seen that $P[O_2, S_2]$ is Σ_p-deterministic (the interleaving of a sequence of Σ_p actions with internal actions of O_2 is uniquely determined by S_2 because it is a deterministic scheduler). Let $P[O_1, S_1]$ be the LTS representation of the set of executions of P with O_1 and consistent with S_1. Since $T(P[O_1, S_1])|\Sigma_p \subseteq T(P[O_2, S_2])|\Sigma_p$,[7] we get that there exists a Σ_p-forward simulation F_{S_1,S_2} from $P[O_1, S_1]$ to $P[O_2, S_2]$. Such a forward simulation defines a relation between states of O_1 and O_2, respectively, by removing the program state, i.e., s_1 and s_2 are related whenever $((s_0, s_1), (s_0, s_2)) \in F_{S_1,S_2}$. For simplicity, this relation is denoted by F_{S_1,S_2} as well. Because of the $record(a)$ actions in Σ_p, we get that F_{S_1,S_2} is a $(C \cup R)$-forward simulation from $O_1[S_1]$ to O_2. It is easy to check that $F_1 \circ F_{S_1,S_2}$ (where $\circ$ is the usual composition of relations) is a $(C \cup R)$-forward simulation from O_1 to O_2. ◀

The two lemmas above imply that:

▶ **Theorem 8.** *$O_1 \leq_s O_2$ iff there exists a $(C \cup R)$-forward simulation from O_1 to O_2.*

The fact that forward simulations are *necessary* for strong refinement makes it possible to derive in a simple way compositional methods for proving strong refinement. In the following we consider the case of *composed* objects defined as a product of a fixed set of objects, and *parametrized* objects defined from a set of "base" objects which are considered as parameters.

[7] Note that $T(P \times O_i, S_i)$ and $T(P[O_i, S_i])$ with $i \in \{1, 2\}$ denote exactly the same set of traces.

We show that strong refinement is a *local* property, i.e., it holds for composed objects if and only if it holds for individual objects in this composition. As usual, we consider compositions of objects with disjoint states and sets of actions. Indeed, any forward simulation between composed objects can be "projected" to a set of forward simulations that hold between individual objects, and vice versa. We state this result for compositions of two objects, the extension to an arbitrary number of objects is obvious.

▶ **Theorem 9.** *Let O_1 and O_2, resp., O'_1 and O'_2, be two objects over an alphabet Σ, resp., Σ', such that $\Sigma \cap \Sigma' = \varnothing$. Then, $O_1 \times O'_1 \leq_s O_2 \times O'_2$ iff $O_1 \leq_s O_2$ and $O'_1 \leq_s O'_2$.*

Next, we consider the case of parametrized objects whose implementation is parametrized by a set of base objects, e.g., snapshot objects defined from a set of atomic registers. We show that if the parametrized object is a strong refinement of an abstract specification *Spec* assuming that the base objects behave according to their own abstract specifications $Spec_i$, then instantiating any base object with an implementation that is a strong refinement of $Spec_i$ leads to an object which remains a strong refinement of *Spec*. Assuming for simplicity only one base object, a parametrized object O can be formally defined as a product $O = Spec_1 \times C$ where $Spec_1$ is the base object's specification and C is the context in which this object is used to derive the implementation of O [8]. To distinguish parametrization from composition, we use $O(Spec_1)$ to denote an object parametrized by a base object $Spec_1$. The next result is an immediate consequence of the fact that the forward simulation admitted by the base object can be composed [9] with the one admitted by the parametrized object (assuming base object's specification) to derive a forward simulation for the instantiation.

▶ **Theorem 10.** *If $O(Spec_1) \leq_s Spec$ and $B_1 \leq_s Spec_1$, then $O(B_1) \leq_s Spec$.*

Finally, it can be shown that the existence of forward simulations is equivalent to *strong linearizability* [14] when concrete objects are related to *atomic* abstract objects. Thus, let O_2 be an atomic object defined by a set of sequential histories *Seq*, i.e., $H(O_2) = \{h : \exists h' \in Seq.\ h \sqsubseteq h'\}$ (according to the definition in Section 3). We say that an object O_1 is *strongly linearizable* w.r.t. O_2, written $O_1 \sqsubseteq_s O_2$, when there exists a function $f : T(O_1) \to Seq$ such that (1) for any trace $\tau \in T(O_1)$, $hist(\tau) \sqsubseteq f(\tau)$, and (2) f is prefix-preserving, i.e., for any two traces $\tau_1, \tau_2 \in T(O_1)$ such that τ_1 is a prefix of τ_2, $f(\tau_1)$ is a prefix of $f(\tau_2)$. It can be shown that the function f induces a forward simulation and vice-versa.

The easier direction is showing that existence of a forward simulation F implies strong linearizability. Essentially, the sequential history associated to a given trace τ (by the function f) is extracted from the atomic object state related by F with the end state of τ.

▶ **Lemma 11.** *Let O_1 be an object and O_2 an atomic object. If there exists a $(C \cup R)$-forward simulation from O_1 to O_2, then O_1 is strongly linearizable w.r.t. O_2.*

Proof. Let F be a $(C \cup R)$-forward simulation from O_1 to O_2. Also, let $state(\tau)$ denote the state of O_1 reached after a trace τ (since O_1 is deterministic, this state is unique). We define a function $f : T(O_1) \to Seq$ by $f(\tau) = h_s$ where h_s satisfies $(state(\tau), (h, h_s)) \in F$. The fact that $hist(\tau) \sqsubseteq f(\tau)$ for every trace τ follows from the definition since (h, h_s) is a valid state of O_2 and $h = hist(\tau)$ (because F preserves call and return actions). The fact that f is prefix-preserving follows from the fact that F is a forward simulation. ◀

[8] For a parametrized object $O = Spec_1 \times C$, the alphabets of $Spec_1$ and C share the call/return actions of $Spec_1$ (the base object) and the alphabet of C contains the call/return actions of O. This is different from the the composition of two objects $O_1 \times O'_1$ where the alphabets of O_1 and O'_1 are disjoint.

[9] Here, we refer to classical composition of relations.

For the reverse direction, strong linearizability implies the existence of a forward simulation where a "concrete" object state s_1 is related to an atomic object state which contains the sequential history associated by the function f witnessing strong linearizability to a trace leading to s_1.

▶ **Lemma 12.** *Let O_1 be an object and O_2 an atomic object. If O_1 is strongly linearizable w.r.t. O_2, then there exists a $(C \cup R)$-forward simulation from O_1 to O_2.*

Proof. Let F be a relation between states of O_1 and O_2 defined by $(s_1, s_2) \in F$ iff there exists a trace τ such that $s_1 = state(\tau)$ and $s_2 = (hist(\tau), f(\tau))$ (by the definition of f in strong linearizability, the latter is a valid state of O_2).

We show that F is a $(C \cup R)$-forward simulation from O_1 to O_2. The fact that it relates the initial object state s_0^1 and the initial state (ϵ, ϵ) of O_2 is trivial. Now, let $s_1, s_1' \in Q_1$, $a \in \Sigma_1$, and $s_2 \in Q_2$, such that $(s_1, a, s_1') \in \delta_1$ and $(s_1, s_2) \in F$. We have to show that there exists $s_2' \in Q_2$ and $\sigma \in \Sigma_2$ such that $(s_1', s_2') \in F$, $s_2 \xrightarrow{\sigma}_{O_2} s_2'$, and $\sigma|(C \cup R) = a|(C \cup R)$. (Let τ be a trace such that $s_1 = state(\tau)$ and $s_2 = (hist(\tau), f(\tau))$. Then, $s_1' = state(\tau \cdot a)$ and $f(\tau)$ is a prefix of $f(\tau \cdot a)$. Several cases are to be discussed:

- if $a \in C$, then $hist(\tau \cdot a) = hist(\tau) \cdot a$ and $hist(\tau) \cdot a \sqsubseteq f(\tau)$ provided that $hist(\tau) \sqsubseteq f(\tau)$. Therefore, $s_2 \xrightarrow{a}_{O_2} (hist(\tau) \cdot a, f(\tau))$ and $(s_1', (hist(\tau) \cdot a, f(\tau))) \in F$.
- if $a \in R$ and the operation identifier k in a occurs in $f(\tau)$, then $s_2 \xrightarrow{a}_{O_2} (hist(\tau) \cdot a, f(\tau))$ and $(s_1', (hist(\tau) \cdot a, f(\tau))) \in F$ like above. If k does not occur in $f(\tau)$, then $f(\tau \cdot a) = f(\tau) \cdot c \cdot a$ where c is the call action corresponding to a (otherwise, $f(\tau \cdot a)$ would not be a linearization of $hist(\tau \cdot a)$). By the definition of O_2, we have that $s_2 \xrightarrow{lin(k) \cdot a}_{O_2} (hist(\tau) \cdot a, f(\tau \cdot a))$ which concludes the proof of this case.
- if $a \notin C \cup R$, then $f(\tau \cdot a)$ is obtained from $f(\tau)$ by appending some sequence of operations with identifiers $k_1, \ldots, k_n$ (this follows from the fact that f is prefix-preserving). Then, $s_2 \xrightarrow{lin(k_1) \cdot \ldots \cdot lin(k_n)}_{O_2} (hist(\tau), f(\tau \cdot a))$ and $(s_1', (hist(\tau), f(\tau \cdot a))) \in F$ (because in this case, $hist(\tau \cdot a) = hist(\tau)$). ◀

Lemma 11 and Lemma 12 imply the following.

▶ **Theorem 13.** *If O_2 is atomic, then $O_1 \sqsubseteq_s O_2$ iff there exists a $(C \cup R)$-forward simulation from O_1 to O_2.*

6 Strong Observational Refinements of Non-Atomic Specifications

We demonstrate that many concurrent objects defined in the literature are strong observational refinements of much simpler abstract objects, even though not necessarily atomic (w.r.t. the definition of atomic object in Section 3). We focus on objects which are not strongly linearizable, since by Theorem 13, the latter are strong refinements of atomic objects.

Figure 3 lists an implementation of a snapshot object with two methods `update(i,data)` for writing the value `data` to a location `i` of a shared array `mem`, and `scan()` for returning a snapshot of the array `mem`.[10] While the implementation of `update` is obvious, a `scan` operation performs several "collect" phases, where it reads successively all the cells of `mem`, until two consecutive phases return the same array.

[10] This is a simplified version of the snapshot object defined by Afek et al. [1].

```
1  procedure update(i,data)
2    mem[i] = data;
3
4  procedure scan()
5    for i = 1 to n do r1[i] = mem[i];
6    repeat
7      r2 = r1;
8      for i = 1 to n do r1[i] = mem[i];
9    until r1 == r2
10   return r1;
```

```
1  procedure update(i,data)
2    mem[i] = data;
3
4  procedure scan()
5    while ( nondet )
6      r = atomic_snapshot();
7      snaps = snaps · r;
8    return r1 ∈ snaps;
```

Figure 3 A snapshot object (on the left), and a concurrent specification (on the right). The shared state of both is an array `mem` of size `n`. The local variables `r1`, `r2`, and `r` are arrays of size `n` (initialized to the same value as `mem`). The local variable `snaps` is a sequence of arrays of size `n` (· denotes the concatenation operator), initially containing a single array which equals the initial value of `mem`. The use of `nondet` means that the loop is executed for an arbitrary number of times. The procedure `atomic_snapshot` returns a snapshot of `mem` in a single step executed in isolation.

This object does *not* admit a forward simulation towards the standard atomic specification where `scan` takes a *single* instantaneous snapshot of the entire array which is subsequently returned (it is not a strong refinement of such a specification). Intuitively, this holds because the linearization point of `scan` depends on future steps in the execution, e.g., a read in the second `for` loop is a linearization point only if it is not followed by updates on array cells before and after the current loop index. This is exactly the scenario in which backward simulations are necessary, intuitively, reading an execution backwards it is possible to identify precisely the linearization points of `scan` invocations. The impossibility of defining such a forward simulation is also a consequence of this object not being strongly linearizable [14].

However, this object is a strong refinement of the simpler "concurrent" specification given on the right of Figure 3 (see more explanation in the full version). The implementation of `update` remains the same, while a `scan` operation performs a sequence of *instantaneous* snapshots of the entire array `mem` and returns *any* snapshot in this sequence. Compared to the implementation on the left, it is simpler because it does not allow that reading the array `mem` is interleaved with other operations. However, it is not atomic since an execution of `scan` contains more than one step. In comparison with the atomic specification, the sequence of snapshots in `scan` allows that an adversary (scheduler) decides on the return value "lazily" after observing other invocations, e.g., updates, exactly as in the concrete implementation. Therefore, the abstract specification in Figure 3 can be used while reasoning about hyperproperties of clients, which is not the case for the atomic specification.

Beyond snapshot objects, Bouajjani et al. [6] show that a similar simplification holds even for concurrent queues and stacks which are not strongly linearizable, e.g., Herlihy&Wing queue [16] and Time-Stamped Stack [11]. These objects admit forward simulations towards "concurrent" specifications where roughly, the elements are stored in a partially-ordered set instead of a sequence (which is consistent with the real-time order between the enqueues/-pushes that added those elements). The enqueues/pushes have no internal steps, while the dequeues/pops have a single internal step which roughly, corresponds to a linearization point that extracts a minimal (for queues) or maximal (for stacks) element from the partially-ordered set. The stack of Afek at al. [2] can also be proved to be a strong refinement of such a specification. These forward simulations imply that these objects are strong refinements of their specifications.

7 Related Work and Discussion

An important contribution of our paper is to put the work on strong linearizability [10, 14, 15] in the context of standard results concerning hyperproperties [8, 9] and property-preserving refinements [3, 17, 18]. McLean [18] showed that refinements do not preserve security properties, which were later found to be instances of the more generic notion of hyperproperty [9]. By exploiting the equivalence between linearizability and refinement [5, 12], our paper clarifies that a stronger notion of linearizability is needed because standard linearizability does not preserve hyperproperties.

Our notion of strong observational refinement is a variation of the hyperproperty-preserving refinement introduced in [9], which takes into account the specificities of concurrent object clients. The relationship between forward simulations and preservation of hyperproperties has been investigated in [3]. They show that the existence of forward simulations is *sufficient* for preserving some specific class of hyperproperties (information-flow security properties like non-interference), corresponding to the straightforward direction of Theorem 8 (Lemma 6); they also show that their condition is *not necessary* in their context. In contrast, our work shows that the existence of forward simulations is *both necessary and sufficient* for preserving any hyperproperty in the context of concurrent object clients.

An important consequence of our results is that strong linearizability is equivalent to the existence of a forward simulation towards an atomic specification. This equivalence has been established independently by Rady [20], albeit using a different formalism that leads to a relatively more complex proof. The equivalence to the well-studied notion of forward simulation immediately implies methods for composing concurrent objects, in particular, *locality* and *instantiation*. This stands in contrast to the effort needed to prove similar results in [14] and [19].

While [14] relates strong linearizability to preserving a rather unspecified class of properties of randomized programs when replacing objects by their atomic specifications, the equivalence we prove implies that strong linearizability is necessary and sufficient for preserving hyperproperties in this context. Note that forward simulations are more general than strong linearizability. Section 6 presents several objects which are *not* strongly linearizable, but which admit forward simulations towards non-atomic abstract specifications. Our results imply that it is sound to use such specifications when reasoning about hyperproperties of client programs. Moreover, as opposed to strong linearizability, forward simulations are applicable to *interval-linearizable* objects [7], which do not have any atomic specification, but are essentially LTSs as in our formalization.

Finally, Bouajjani et al. [6] show that intricate implementations of concurrent stacks and queues like Herlihy&Wing queue [16] and Time-Stamped Stack [11] admit forward simulations towards non-atomic abstract specifications, but they do not discuss the connection between existence of forward simulations and preservation of hyperproperties, which is the main contribution of our paper.

Our definition of strong observational refinement and its relation to forward simulations deepens our understanding of the role of strong linearizability in preserving hyperproperties. We plan to explore strong observational refinement of almost-atomic objects and develop additional proof methodologies. Also, our notion of strong refinement uses deterministic schedulers that model strong adversaries w.r.t. Aspnes' classification [4], and it is interesting to explore variations of this notion that take into account other adversary models.

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. Yehuda Afek, Eli Gafni, and Adam Morrison. Common2 extended to stacks and unbounded concurrency. *Distributed Computing*, 20(4):239–252, 2007. doi:10.1007/s00446-007-0023-3.
3. Rajeev Alur, Pavol Cerný, and Steve Zdancewic. Preserving Secrecy Under Refinement. In Michele Bugliesi, Bart Preneel, Vladimiro Sassone, and Ingo Wegener, editors, *Automata, Languages and Programming, 33rd International Colloquium, ICALP 2006, Venice, Italy, July 10-14, 2006, Proceedings, Part II*, volume 4052 of *Lecture Notes in Computer Science*, pages 107–118. Springer, 2006. doi:10.1007/11787006_10.
4. James Aspnes. Randomized protocols for asynchronous consensus. *Distributed Computing*, 16(2-3):165–175, 2003. doi:10.1007/s00446-002-0081-5.
5. Ahmed Bouajjani, Michael Emmi, Constantin Enea, and Jad Hamza. Tractable Refinement Checking for Concurrent Objects. In Sriram K. Rajamani and David Walker, editors, *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2015, Mumbai, India, January 15-17, 2015*, pages 651–662. ACM, 2015. doi:10.1145/2676726.2677002.
6. Ahmed Bouajjani, Michael Emmi, Constantin Enea, and Suha Orhun Mutluergil. Proving Linearizability Using Forward Simulations. In Rupak Majumdar and Viktor Kuncak, editors, *Computer Aided Verification - 29th International Conference, CAV 2017, Heidelberg, Germany, July 24-28, 2017, Proceedings, Part II*, volume 10427 of *Lecture Notes in Computer Science*, pages 542–563. Springer, 2017. doi:10.1007/978-3-319-63390-9_28.
7. Armando Castañeda, Sergio Rajsbaum, and Michel Raynal. Unifying Concurrent Objects and Distributed Tasks: Interval-Linearizability. *J. ACM*, 65(6):45:1–45:42, 2018. doi:10.1145/3266457.
8. Michael R. Clarkson, Bernd Finkbeiner, Masoud Koleini, Kristopher K. Micinski, Markus N. Rabe, and César Sánchez. Temporal Logics for Hyperproperties. In Martín Abadi and Steve Kremer, editors, *Principles of Security and Trust - Third International Conference, POST 2014, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2014, Grenoble, France, April 5-13, 2014, Proceedings*, volume 8414 of *Lecture Notes in Computer Science*, pages 265–284. Springer, 2014. doi:10.1007/978-3-642-54792-8_15.
9. Michael R. Clarkson and Fred B. Schneider. Hyperproperties. *Journal of Computer Security*, 18(6):1157–1210, 2010. doi:10.3233/JCS-2009-0393.
10. Oksana Denysyuk and Philipp Woelfel. Wait-Freedom is Harder Than Lock-Freedom Under Strong Linearizability. In Yoram Moses, editor, *Distributed Computing - 29th International Symposium, DISC 2015, Tokyo, Japan, October 7-9, 2015, Proceedings*, volume 9363 of *Lecture Notes in Computer Science*, pages 60–74. Springer, 2015. doi:10.1007/978-3-662-48653-5_5.
11. Mike Dodds, Andreas Haas, and Christoph M. Kirsch. A Scalable, Correct Time-Stamped Stack. In Sriram K. Rajamani and David Walker, editors, *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2015, Mumbai, India, January 15-17, 2015*, pages 233–246. ACM, 2015. doi:10.1145/2676726.2676963.
12. Ivana Filipovic, Peter W. O'Hearn, Noam Rinetzky, and Hongseok Yang. Abstraction for concurrent objects. *Theor. Comput. Sci.*, 411(51-52):4379–4398, 2010. doi:10.1016/j.tcs.2010.09.021.
13. Joseph A. Goguen and José Meseguer. Security Policies and Security Models. In *1982 IEEE Symposium on Security and Privacy, Oakland, CA, USA, April 26-28, 1982*, pages 11–20. IEEE Computer Society, 1982. doi:10.1109/SP.1982.10014.
14. Wojciech M. Golab, Lisa Higham, and Philipp Woelfel. Linearizable implementations do not suffice for randomized distributed computation. In Lance Fortnow and Salil P. Vadhan, editors, *Proceedings of the 43rd ACM Symposium on Theory of Computing, STOC 2011, San Jose, CA, USA, 6-8 June 2011*, pages 373–382. ACM, 2011. doi:10.1145/1993636.1993687.

15 Maryam Helmi, Lisa Higham, and Philipp Woelfel. Strongly linearizable implementations: possibilities and impossibilities. In Darek Kowalski and Alessandro Panconesi, editors, *ACM Symposium on Principles of Distributed Computing, PODC '12, Funchal, Madeira, Portugal, July 16-18, 2012*, pages 385–394. ACM, 2012. doi:10.1145/2332432.2332508.

16 Maurice Herlihy and Jeannette M. Wing. Linearizability: A Correctness Condition for Concurrent Objects. *ACM Trans. Program. Lang. Syst.*, 12(3):463–492, 1990. doi:10.1145/78969.78972.

17 Nancy A. Lynch and Frits W. Vaandrager. Forward and Backward Simulations: I. Untimed Systems. *Inf. Comput.*, 121(2):214–233, 1995. doi:10.1006/inco.1995.1134.

18 John McLean. A general theory of composition for trace sets closed under selective interleaving functions. In *1994 IEEE Computer Society Symposium on Research in Security and Privacy, Oakland, CA, USA, May 16-18, 1994*, pages 79–93. IEEE Computer Society, 1994. doi:10.1109/RISP.1994.296590.

19 Sean Ovens and Philipp Woelfel. Strongly Linearizable Implementations of Snapshots and Other Types. In *38th ACM Symposium on Principles of Distributed Computing (PODC 2019)*, 2019.

20 Amgad Sadek Rady. Characterizing Implementations that Preserve Properties of Concurrent Randomized Algorithms. Master's thesis, York University, Toronto, Canada, 2017.

21 Gerhard Schellhorn, Heike Wehrheim, and John Derrick. How to Prove Algorithms Linearisable. In P. Madhusudan and Sanjit A. Seshia, editors, *Computer Aided Verification - 24th International Conference, CAV 2012, Berkeley, CA, USA, July 7-13, 2012 Proceedings*, volume 7358 of *Lecture Notes in Computer Science*, pages 243–259. Springer, 2012. doi:10.1007/978-3-642-31424-7_21.