Theoretical Computer Science 411 (2010) 4379–4398

Contents lists available at ScienceDirect

Theoretical Computer Science

journal homepage: www.elsevier.com/locate/tcs

Abstraction for concurrent objects

Ivana Filipović, Peter O'Hearn, Noam Rinetzky [*], Hongseok Yang

Queen Mary University of London, UK

ARTICLE INFO

Keywords:
Linearizability
Sequential consistency
Observational equivalence
Observational refinement

ABSTRACT

Concurrent data structures are usually designed to satisfy correctness conditions such as sequential consistency or linearizability. In this paper, we consider the following fundamental question: What guarantees are provided by these conditions for client programs? We formally show that these conditions can be *characterized* in terms of observational refinement. Our study also provides a new understanding of sequential consistency and linearizability in terms of abstraction of dependency between computation steps of client programs.

© 2010 Elsevier B.V. All rights reserved.

1. Introduction

The design and implementation of correct and efficient concurrent programs is a challenging problem. Thus, when developing a critical concurrent application, programmers often prefer to reuse ready-made highly optimized concurrent data structures that have been designed, implemented, and verified by experts, rather then implementing these data structures by themselves. Unfortunately, there is an important gap in our theoretical understanding here: Application programmers trained in formal methods justify the integration of a ready-made data structure into their system, by expecting the data structure to guarantee that it will not lead programs which use it to behave in ways which cannot be inferred from its interface. However, ready-made concurrent data structures are developed to guarantee correctness conditions proposed by researchers in concurrent programming. Surprisingly, there are no known formal connections between the expectations of the users of concurrent data structures and the guarantees made by their providers. Bridging this gap is the aim of this paper.

We consider a program to be comprised of a (possibly multithreaded) application-specific *client program* and a concurrent data structure. Usually, programmers expect the externally observable behaviour of their program not to change if they replace an inefficient, but obviously correct, implementation of a data structure with a highly optimized one. In the programming language community, this expectation has been formalized as *observational refinement* [10,14,21]. Informally, an implementation O of a data structure is an observational refinement of an implementation O' of the same data structure, if every observable behaviour of *any client program* using O can also be observed when the program uses O' instead. In this paper, for example, the observable behaviour means the contents of the parts of final states that are not related to the data structure implementation, e.g., the local variables or global variables defined in the client program.

On the other hand, concurrent data structures are designed to fulfil correctness conditions that were proposed by the concurrent-algorithm community, e.g., *sequential consistency* [17] or *linearizability* [12]. Informally, both conditions prescribe an "atomic flavour" to the implementation of the data structure's methods: Sequential consistency requires that for every sequence of possibly overlapping method invocations by threads, there exists a sequence of non-overlapping

[*] Corresponding address: Department of Computer Science, Queen Mary University of London, London E1 4NS, UK. Tel.: +44 0 20 7882 5237; fax: +44 0 20 8980 6533.

E-mail addresses: ivanam@dcs.qmul.ac.uk (I. Filipović), ohearn@dcs.qmul.ac.uk (P. O'Hearn), maon@dcs.qmul.ac.uk (N. Rinetzky), hyang@dcs.qmul.ac.uk (H. Yang).

0304-3975/$ – see front matter © 2010 Elsevier B.V. All rights reserved.
doi:10.1016/j.tcs.2010.09.021

invocations, where each thread makes the same sequence of calls and receives the same sequence of responses as in the original sequence. Linearizability adds to the above requirement that the order of non-overlapping invocations in the former (original) sequence be preserved in the latter one. Note that, formally, the definitions of neither sequential consistency nor linearizability provide any guarantees regarding the behaviour of client code that uses a sequentially consistent or a linearizable data structure.

The difference between the two viewpoints leads to the following natural question: Do the correctness conditions that are used in the design and implementation of concurrent data structures, e.g., sequential consistency or linearizability, imply observational refinement, which formalizes the expectations of the application programmer who use these data structures? In this paper, we give the first systematic answer (as far as we are aware) to this question.

This paper contains three main results, which are stated more formally in Section 3. Informally, the first result is that, in general, linearizability coincides with observational refinement. The second result considers a special class of programs where the communication between threads is done only through invocations of methods of the concurrent data structure (and thus, in a sense, exposed to the data structure). In this case, our result says, sequential consistency is equivalent to observational refinement. The third result focuses on the notions/definitions of linearizability and sequential consistency and provides a formal methods/programming language view of these notions. More specifically, it shows how the definitions of linearizability and sequential consistency can be understood in terms of the abstraction of all possible dependences between computation steps.

For programmers using concurrent data structures, our first two results pinpoint when it is possible to replace a data structure by another sequentially consistent or linearizable one in their (client) programs, while preserving observable properties of the programs. The *soundness* direction in this connection (that linearizability or, for our special class of programs, sequential consistency implies observational refinement) has been folklore amongst concurrent-algorithm researchers, and our results provide the first formal confirmation of this folklore. On the other hand, as far as we know, the *completeness* direction (when observational refinement implies linearizability or sequential consistency) is not prefigured or otherwise suggested in the literature.

For programmers implementing concurrent data structures, our third result provides a new understanding of these correctness conditions—they are conservative over-approximations of the dependencies between method invocations that may arise due to operations from a (client) program using the data structure.

Outline. In Section 2, we start with an informal description of our mathematical model and the notions of linearizability, sequential consistency and observational refinement through a series of examples. In Section 3, we state our main results regarding the relationship between correctness conditions and observational refinement. In Section 4, we describe the formal setting, such as syntax and semantics of the programming language, and in Section 5.1, we prove our main results in this setting. In Section 6, we revisit the definitions of sequential consistency and linearizability, and provide an analysis of them in terms of the dependency between computation steps. Section 7 contains a discussion of related work, and Section 8 concludes.

Note. An extended abstract of this paper was published in the proceedings of the 18th European Symposium on Programming (ESOP'09). This paper includes proofs that were omitted from the conference version, new examples, and an extension of the results of the conference paper to incomplete executions.

2. Background

This section provides an overview of the programming model that we use in the paper, and explains the notions of *linearizability*, *sequential consistency* and *observational refinement*. The presentation is at a semi-technical level; a more formal treatment of this material is presented in later sections of the paper.

2.1. Programs, object systems and histories

A concurrent data structure provides a set of procedures, which may be invoked by concurrently executing threads of a client program using the data structure. Thus, procedure invocations may overlap. In our setting, we assume that a concurrent data structure neither creates threads nor calls a procedure of a client program, a standard assumption used in the concurrent-algorithm community. We refer to a collection of concurrent data structures as an *object system*.

In this paper, we are not interested in the implementation of an object system; we are only interested in the possible interactions between an object system and a client program. Thus, we assume that an object system is represented by a set of *histories*. Every history records a possible interaction between the object system and a client application program. The interaction is given in the form of sequences of procedure invocations made by the client and the responses which it receives. A client program can use an object system only by interacting with it according to one of the object system's histories.[1]

[1] This is a standard assumption in concurrent-algorithm work, which Herlihy and Shavit refer to as *interference freedom* [11]: it is an assumption that would have to be verified by other means when applying the theory to particular programming languages or programs. See also Section 7.

H_0	H_1	H_2	H_3
$(t_1, call\, q.enq(1));$	$(t_1, call\, q.enq(1));$	$(t_2, call\, q.enq(2));$	$(t_1, call\, q.enq(1));$
$(t_1, ret()\, q.enq);$	$(t_1, ret()\, q.enq);$	$(t_2, ret()\, q.enq);$	$(t_2, call\, q.enq(2));$
$(t_2, call\, q.deq());$	$(t_2, call\, q.enq(2));$	$(t_1, call\, q.enq(1));$	$(t_1, ret()\, q.enq);$
$(t_2, ret(1)\, q.deq)$	$(t_2, ret()\, q.enq)$	$(t_1, ret()\, q.enq)$	$(t_2, ret()\, q.enq)$

Fig. 1. Example histories. Histories are written top down. We use indentation to help distinguish visually between actions made by thread t_1 and actions made by thread t_2.

τ^1	τ^2	τ
$(t_1, x{:=}11);$	$(t_2, y{:=}x);$	$(t_1, x{:=}11);$
$(t_1, call\, q.enq(1));$	$(t_2, call\, q.enq(2));$	$(t_2, y{:=}x);$
$(t_1, ret()\, q.enq);$	$(t_2, ret()\, q.enq);$	$(t_1, call\, q.enq(1));$
$(t_1, x{:=}10)$	$(t_2, y{:=}x);$	$(t_2, call\, q.enq(2));$
	$(t_2, x{:=}0)$	$(t_1, ret()\, q.enq);$
		$(t_1, x{:=}10);$
		$(t_2, ret()\, q.enq);$
		$(t_2, y{:=}x);$
		$(t_2, x{:=}0)$

Fig. 2. Example traces. Traces are written top down. We use indentation to help distinguish visually between actions made by thread t_1 and actions made by thread t_2.

Example 1. The history H_0, shown in Fig. 1, records an interaction in which thread t_1 first enqueues 1 into the queue q and then thread t_2 dequeues 1 from the queue. □

Example 2. The histories H_1, H_2, and H_3, shown in Fig. 1, record interactions in which thread t_1 enqueues 1 and thread t_2 enqueues 2, both to the same queue. In H_1, the invocation made by t_1 happens before that of t_2 (i.e., t_1 gets a response before t_2 invokes its own procedure). In H_2, it is the other way around. In H_3, the two invocations overlap. □

While a history describes only the interaction of a client program with an object system, a trace represents an entire computation of the client program.

Example 3. Trace τ^1 in Fig. 2 shows a sequence of actions performed by the thread t_1, which interacts with a concurrent queue object. The first and the last actions are assignments to variables, which are independent of the queue object. The other two actions, on the other hand, record the interaction between the client and the queue. They express that t_1 enqueues 1 into queue q. Concretely, $(t_1, call\, q.enq(1))$ denotes a call of this enqueue operation and $(t_1, ret()\, q.enq)$ the response of the operation. Note that the trace does not express the internal steps of the implementation of the queue object. If we had not hidden these steps, the trace would have had object-internal actions by the thread t_1 between the call and return actions.

Trace τ^1 can be parallel-composed (i.e., interleaved) with traces of other threads, resulting in traces for concurrent execution of multiple threads. Trace τ, shown in Fig. 2, is an example of one such interleaving, where τ^1 is parallel-composed with trace τ^2 of thread t_2, also given in Fig. 2, in which t_2 enqueues the value 2. Note that by overlapping the calls and returns of two enqueue operations, trace τ expresses that the queue object named q is accessed concurrently by t_1 and t_2. Also, notice that the interaction between these two threads and the queue object is captured precisely by the history H_3 in Fig. 1. □

2.2. Sequential consistency and linearizability

Informally, an object system OS_C is *sequentially consistent* with respect to an object system OS_A if for every history H_C in OS_C, there exists a history H_A in OS_A that is just another interleaving of threads' actions in H_C: in both H_C and H_A, the same threads invoke the same sequences of operations (i.e., procedure invocations) and receive the same sequences of responses. We say that such H_C and H_A are *weakly equivalent*. We use the term *weak* equivalence to emphasize that the only relation between H_C and H_A is that they are different interleavings of the same sequential threads. OS_C is *linearizable* with respect to OS_A if for every history H_C in OS_C, there is some H_A in OS_A such that (i) H_C and H_A are weakly equivalent and (ii) the order of non-overlapping procedure invocations of H_C is preserved in H_A.[2] In the context of this paper, the main difference between sequential consistency and linearizability is, intuitively, that the former preserves the order of non-overlapping invocations of the *same* thread only, not the order of invocations by different threads, while the latter preserves the order of *all* the non-overlapping operations by *all* threads.

In both sequential consistency and linearizability, OS_A is a part of a specification, describing the expected behaviour of data structures. Hence, the weak equivalence between OS_C and OS_A usually implies that OS_C shows the expected behaviour

[2] It is common to require that OS_A be comprised of sequential histories, i.e., ones in which invocations do not overlap. In this setting, linearizability intuitively means that every operation appears to happen instantaneously between its invocation and its response. This requirement is not technically necessary for our results, so we do not impose it. For details, see Section 7.

τ^{1a}	τ^{2a}
$(t_1, \mathit{call}\, q.\mathit{enq}(1));$	$(t_2, \texttt{assume}(x = 1));$
$(t_1, \mathit{ret}()\, q.\mathit{enq});$	$(t_2, \mathit{call}\, q.\mathit{enq}(2));$
$(t_1, x := 1);$	$(t_2, \mathit{ret}()\, q.\mathit{enq})$
(t_1, skip)	

$\tau^{a'}$	$\tau^{a''}$	$\tau^{a'''}$	$\tau^{a''''}$
$(t_1, \mathit{call}\, q.\mathit{enq}(1));$	$(t_1, \mathit{call}\, q.\mathit{enq}(1));$	$(t_1, \mathit{call}\, q.\mathit{enq}(1));$	$(t_1, \mathit{call}\, q.\mathit{enq}(1));$
$(t_1, \mathit{ret}()\, q.\mathit{enq});$	$(t_1, \mathit{ret}()\, q.\mathit{enq});$	$(t_1, \mathit{ret}()\, q.\mathit{enq});$	$(t_1, \mathit{ret}()\, q.\mathit{enq});$
$(t_1, x := 1);$	$(t_1, x := 1);$	$(t_1, x := 1);$	$(t_1, x := 1);$
$(t_1, \mathit{skip});$	$(t_2, \texttt{assume}(x = 1));$	$(t_2, \texttt{assume}(x = 1));$	$(t_2, \texttt{assume}(x = 1));$
$(t_2, \texttt{assume}(x = 1));$	$(t_1, \mathit{skip});$	$(t_2, \mathit{call}\, q.\mathit{enq}(2));$	$(t_2, \mathit{call}\, q.\mathit{enq}(2));$
$(t_2, \mathit{call}\, q.\mathit{enq}(2));$	$(t_2, \mathit{call}\, q.\mathit{enq}(2));$	$(t_1, \mathit{skip});$	$(t_2, \mathit{ret}()\, q.\mathit{enq});$
$(t_2, \mathit{ret}()\, q.\mathit{enq})$	$(t_2, \mathit{ret}()\, q.\mathit{enq})$	$(t_2, \mathit{ret}()\, q.\mathit{enq})$	(t_1, skip)

Fig. 3. Example traces. Traces are written top down. We use indentation to help distinguish visually between actions made by thread t_1 and actions made by thread t_2.

of data structures with respect to the order of method invocations. For example, if a concurrent queue OS_C is sequentially consistent or linearizable with respect to the standard sequential queue OS_A, every enqueue of a value in the concurrent queue must take effect before the dequeue of the value.

Example 4. The histories H_1, H_2 and H_3 in Fig. 1 are weakly equivalent. But none of them is weakly equivalent to H_0 in the same figure. □

Example 5. The histories H_1, H_2, and H_3, shown in Fig. 1, record different interactions between the concurrent queue object and a client program, which may result from, e.g., different interleavings of traces τ^1 and τ^2 in Fig. 2. However, no interleaving of τ^1 and τ^2 may produce the interaction recorded by history H_0 in Fig. 1. □

Example 6. The history H_3 in Fig. 1 is linearizable with respect to H_1 *and* with respect to H_2, because H_3 does not have non-overlapping invocations.

On the other hand, H_1 is not linearizable with respect to H_2; in H_1, the enqueue of t_1 is completed before that of t_2 even starts, but this order on these two enqueues is reversed in H_2. □

The next example shows one consequence of history H_1, shown in Fig. 1, not being linearizable with respect to H_2, which also appears in the same figure.

Example 7. Consider all possible interleavings of traces τ^{1a} and τ^{2a} in Fig. 3. In trace τ^{1a}, thread t_1 enqueues 1 and then sets the global variable x to value 1. In trace τ^{2a}, thread t_2 waits until $\texttt{assume}(x = 1)$ is true, i.e., the global variable x has the value 1. If x never becomes 1, thread t_2 gets stuck in this waiting state. Assume that x is initialized to 0. Then, in every interleaved trace τ^a between τ^{1a} and τ^{2a}, if the execution of τ^a proceeds without t_2 getting stuck, the enqueue by t_1 should be completed before the enqueue by t_2. This means that in all such non-stuck traces, the interaction with the queue is H_1, not H_2, although H_1 and H_2 are weakly equivalent. In fact, traces $\tau^{a'}$, $\tau^{a''}$, $\tau^{a'''}$ and $\tau^{a''''}$, shown in Fig. 3, are the only interleavings of τ^{1a} and τ^{2a} where thread t_2 does not get stuck. □

Example 7 illustrates that if histories are not related according to the notion of linearizability, they can (sometimes) be distinguished by client programs, especially when their threads synchronize without using methods from an object system; for instance, the threads in Example 7 synchronize using the global variable x. Our results in later sections imply that "sometimes" here is indeed "always", and that if linearizability holds between two histories, we can never construct a similar example that distinguishes the histories.

2.3. *Observational refinement*

Our notion of observational refinement is based on observing the initial and final values of variables of client programs. (One can think of the program as having a final command "print all variables".) We say that an object system OS_C observationally refines an object system OS_A if for every program P with OS_A, replacing OS_A by OS_C does not generate new observations: for every initial state s, the execution of P with OS_C at s produces only those output states that can already be obtained by running P with OS_A at s.

Example 8. Consider a resource pool data structure, which manages a fixed finite number of resources. Clients of the pool can obtain a resource by invoking a `malloc`-like procedure and can release a resource by invoking a `free`-like procedure. Assume that every resource has a unique identifier and consider the following three implementations of a resource pool: a stack, a queue and a set. In the set implementation, the procedure for resource allocation non-deterministically chooses an available resource in the pool.

If clients of the pool can compare resource identifiers (and thus may behave differently according to the identifier of the resources they obtain), then the stack-based pool and the queue-based pool observationally refine the set-based pool, but neither vice versa nor each other. □

3. Main results

We can now state the main results of this paper, which are the characterization of sequential consistency and linearizability in terms of observational refinement and abstraction:

1. OS_C observationally refines OS_A iff OS_C is sequentially consistent with respect to OS_A, assuming that client operations of different threads are independent (and so they commute). This assumption is met when each thread accesses only thread-local variables or resources in its client operations.
2. OS_C observationally refines OS_A iff OS_C is linearizable with respect to OS_A, assuming that client operations may use at least one shared global variable.[3]
3. We suggest a novel abstraction-based understanding of sequential consistency and linearizability by presenting them as over-approximations of client-induced dependencies.

A note regarding the assumed programming models. Although our technical results are stated for particular programming models, we point out that this is mainly to make our presentation concrete. The results rely on only certain features of the models, and they can be generalized to more general models, as we discuss below.

The essence of the particular class of programs in our soundness result regarding sequential consistency is that all the communication between threads is done through the object system. We captured this condition by requiring that the client operations of different threads be independent and commute, and we have given an example where client operations of each thread access only thread-local variables or resources. A possible generalization is to allow threads to use shared resources, in addition to the object system, that provide independent operation suites to different threads, e.g., a shared log, where the log is not part of the observable behaviour of the system.

The essence of the particular class of programs in our completeness result regarding linearizability is that threads have effective means of communication besides the object system, i.e., there is inter-thread communication that is not expressed in histories of the object system. We captured this condition by providing threads with at least one shared (atomic) integer variable (in addition to the object system) for thread communication. Our completeness result regarding linearizability can be easily adapted to different models, as long as threads can communicate, (in addition to the object system) using at least one shared sequentially consistent resource that has an unbounded number of client-observable states, *e.g.*, a two-state binary semaphore together with a (possibly non-atomic) unbounded integer object.

4. The formal setting

In this section, we describe the formal setting where we will present our results. In Section 4.1, we define the syntax of our programming language. In Sections 4.2 and 4.3, we define the semantics of our programming language: Following Brookes [5], we define the semantics in two stages. In the first stage, which is shown in Section 4.2, we define a trace model, where the traces are built from atomic actions. This model resolves all concurrency by interleaving. In the second stage, which is shown in Section 4.3, we define the evaluation of these action traces with initial states. In Section 4.4, we give a formal definition of observational refinement for object systems.

4.1. The programming language

We assume that we are given a fixed collection O of objects, with method calls $o.f(n)$. For simplicity, all methods will take one integer argument and return an integer value. We will denote method calls by $x{:=}o.f(e)$.

The syntax of sequential commands C and complete programs P is given below:

$$C \ ::= \ \mathtt{c} \mid x{:=}o.f(e) \mid C; C \mid C + C \mid C^{\star} \qquad P \ ::= \ C_1 \parallel \cdots \parallel C_n$$

Here, $\mathtt{c}$ ranges over an unspecified collection $\mathtt{PComm}$ of primitive commands, $+$ is non-deterministic choice, ; is sequential composition, and $(\cdot)^{\star}$ is Kleene-star (iterated ;). We use $+$ and $(\cdot)^{\star}$ instead of conditionals and while loops for theoretical simplicity: given appropriate primitive commands the conditionals and loops can be encoded. In this paper, we assume that the primitive commands include assume statements $\mathtt{assume}(b)$ and assignments $x{:=}e$ not involving method calls.[4]

[3] We assume that the shared global variables are sequentially consistent. However, being part of the client code, the interaction between the client and the shared global variables is not recorded in the object system.

[4] The $\mathtt{assume}(b)$ statement acts as skip when the input state satisfies b. If b does not hold in the input state, the statement deadlocks and *does not produce any output states.*

τ_1^{nwf}	τ_2^{nwf}	τ_3^{nwf}
$(t_2, call\, q.enq(2))$;	$(t_1, call\, q.enq(1))$;	$(t_1, call\, q.enq(1))$;
$(t_1, ret()\, q.enq)$;	$(t_1, call\, q.enq(3))$;	$(t_1, x := 3)$;
$(t_2, ret()\, q.enq)$	$(t_1, ret()\, q.enq)$;	$(t_1, ret()\, q.enq)$
	$(t_1, ret()\, q.enq)$	

Fig. 4. Example non-well-formed traces. Traces are written top down. We use indentation to help distinguish visually between actions made by thread t_1 and actions made by thread t_2.

4.2. The action trace model

In this section, we develop the first stage of our semantics: We define a trace model, where the traces are built from atomic actions, which resolves all concurrency by interleaving.

Definition 9. An **atomic action** (for short, action) φ is a client operation or a call or return action:

$$\varphi \ ::= \ (t, a) \ | \ (t, call\, o.f(n)) \ | \ (t, ret(n)\, o.f).$$

Here, t is a thread-id (i.e., a natural number), a in (t, a) is an atomic client operation taken from an unspecified set Cop_t (parameterized by the thread-id t), and n is an integer. An **action trace** (for short, trace) τ is a finite sequential composition of actions (i.e., $\tau \ ::= \ \varphi; \cdots ; \varphi$).

We identify a special class of traces where calls to object methods run sequentially.

Definition 10. A trace τ is **sequential** when all calls in τ are immediately followed by matching returns, that is, τ belongs to the set

$$\left(\bigcup_{t,a,o,f,n,m} \{ (t, a), (t, call\, o.f(n)); (t, ret(m)\, o.f) \} \right)^* \left(\bigcup_{t,o,f,n} \{ \epsilon, (t, call\, o.f(n)) \} \right).$$

Intuitively, sequentiality means that all method calls to objects run atomically. Sequentiality ensures that method calls and returns are properly matched (possibly except the last call), so, for instance, no sequential traces start with a return action, such as $(t, ret(3)\, o.f)$.

Example 11. Traces τ^1 and τ^2, shown in Fig. 2, are sequential, but Trace τ in the same figure is not. When viewed as traces, histories H_0, H_1 and H_2 in Fig. 1 are sequential, but history H_3 in the same figure is not. □

The execution of a program in this paper generates only well-formed traces.

Definition 12 (*Executing Thread*). The **executing thread** of an action φ, denoted as $\mathsf{getTid}(\varphi)$, is the thread-id (i.e., the first component) of φ.

Definition 13. The **projection of a trace** τ **to thread-id** t, denoted as $\tau|_t$, is the subsequence of τ comprised of the actions executed by thread t.

Definition 14. A trace τ is **well-formed** iff $\tau|_t$ is sequential for all thread-ids t.

The well-formedness condition formalizes two properties of traces. Firstly, it ensures that all the returns should have corresponding method calls. Secondly, it formalizes the intuition that each thread is a sequential program, if it is considered in isolation. Thus, when the thread calls a method $o.f$, it has to wait until the method returns, before doing anything else. We denote the set of all well-formed traces by *WTraces*.

Example 15. All the traces shown in Figs. 2 and 3 are well-formed. The traces shown in Fig. 4 are not well-formed: In trace τ_1^{nwf}, thread t_1 gets a response before it invokes a procedure call on the object system. In trace τ_2^{nwf}, thread t_1 does not wait for the response of the first procedure invocation before it makes a second invocation. In trace τ_3^{nwf}, thread t_1 assigns a value to the global variable x after it invokes a procedure on the object system and before it gets its response. □

Our trace semantics $T(-)$ defines the meaning of sequential commands and programs in terms of traces, and it is shown in Fig. 5. In our model, a sequential command C means a set $T(C)t$ of well-formed traces, which is parametrized by the id t of a thread running the command. The semantics of a complete program (a parallel composition) P, on the other hand, is a non-parametrized set $T(P)$ of well-formed traces; instead of taking thread-ids as parameters, $T(P)$ creates thread-ids.

Two cases of our semantics are slightly unusual and need further explanations. The first case is for the primitive commands c. In this case, the semantics assumes that we are given an interpretation $[\![\mathsf{c}]\!]_t$ of c, where c means finite sequences of atomic client operations (i.e., $[\![\mathsf{c}]\!]_t \subseteq Cop_t^+$). By allowing sequences of length 2 or more, this assumed interpretation allows the possibility that c is not atomic, but implemented by a sequence of atomic operations. The second case is for method calls. Here the semantics distinguishes calls and returns to objects, to be able to account for concurrency (overlapping operations). Given $x{:=}o.f(e)$, the semantics non-deterministically chooses two integers n and n', and uses them

$$\begin{aligned}
T(\mathtt{c})t &= \{\ (t, a_1); (t, a_2); \ldots; (t, a_k) \ \mid\ a_1; a_2; \ldots; a_k \in [\![\mathtt{c}]\!]_t\ \} \\
T(x{:=}o.f(e))t &= \{\ \tau; (t, call\, o.f(n)); (t, ret(n')\, o.f); \tau' \ \mid \\
&\qquad n, n' {\in} Integers \wedge \tau {\in} T(\mathtt{assume}(e{=}n))t \wedge \tau' {\in} T(x{:=}n')t\ \} \\
T(C_1; C_2)t &= \{\ \tau_1; \tau_2 \ \mid\ \tau_i {\in} T(C_i)t\ \} \\
T(C_1{+}C_2)t &= T(C_1)t \cup T(C_2)t \\
T(C^\star)t &= (T(C)t)^\star \\
T(C_1 \parallel \cdots \parallel C_n) &= \bigcup \{\ interleave(\tau_1, \ldots, \tau_n) \ \mid\ \tau_i \in T(C_i)i \wedge 1 \le i \le n\ \}
\end{aligned}$$

Fig. 5. Action trace model. Here $\tau \in interleave(\tau_1, \ldots, \tau_n)$ iff every action in τ is done by a thread $1 \le i \le n$ and $\tau|_i = \tau_i$ for every such thread i.

to describe a call with input n and a return with result n'. In order to ensure that the argument e evaluates to n, the semantics inserts the assume statement $\mathtt{assume}(e{=}n)$ before the call action, and to ensure that x gets the return value n', it adds the assignment $x{:=}n'$ after the return action. Note that some of the choices here might not be feasible; for instance, the chosen n might not be the value of the parameter expression e when the call action is invoked, or the concurrent object never returns n' when called with n. The next evaluation stage of our semantics will filter out all these infeasible call/return pairs.

Lemma 16. *For all sequential commands C, programs P and thread-ids t, both $T(C)t$ and $T(P)$ contain only well-formed traces.*

4.2.1. Object systems

The semantics of objects is given using histories, which are sequences of calls and returns to objects. We first define precisely what the individual elements in the histories are.

Definition 17. An **object action** is a call or return:

$$\psi \ ::=\ (t, call\, o.f(n)) \mid (t, ret(n)\, o.f).$$

A **history** H is a finite sequence of object actions (i.e., $H\ ::=\ \psi; \psi; \ldots; \psi$). If a history H is well-formed when viewed as a trace, we say that H is **well-formed**.

Note that in contrast to traces, histories do not include atomic client operations (t, a). We will use $\mathcal{A}$ for the set of all actions, $\mathcal{A}_o$ for the set of all object actions, and $\mathcal{A}_c$ for $\mathcal{A} - \mathcal{A}_o$, i.e., the set of all client operations.

We follow Herlihy and Wing's approach [12], and define object systems.

Definition 18. An **object system** *OS* is a set of well-formed histories.

Notice that *OS* is a collective notion, defined for all objects together rather than for them independently. Sometimes, the traces of a system satisfy special properties.

Definition 19. The **projection of a history** H **to object** o, denoted as $H|_o$, is the subsequence of H comprised of the object actions (i.e., call and return actions) on object o. The **projection of a trace** τ **to object actions**, denoted as $\mathsf{getHistory}(\tau)$, is the subsequence of τ that consists of all the object actions in τ. The **projection of a trace** τ **to client actions**, denoted as $\mathsf{getClient}(\tau)$, is the subsequence of τ comprised of the atomic client operations in τ.

Example 20. Consider the history H_3 in Fig. 1. The only object that appears in history H_3 is q. Thus, the projection of H_3 on object q is $H_3|_q = H_3$, and for objects $q' \neq q$, we have that $H_3|_{q'} = \epsilon$. The history H_3 is the result of projecting trace τ in Fig. 2 on object actions: $\mathsf{getHistory}(\tau) = H_3$. □

Definition 21. Let *OS* be an object system. We say that *OS* is **sequential** iff it contains only sequential traces; *OS* is **local** iff for any well-formed history H, $H \in OS \iff (\forall o. H|_o \in OS)$.

A local object system is one in which the set of histories for all the objects together is determined by the set of histories for each object individually. Intuitively, locality means that objects can be specified in isolation. Sequential and local object systems are commonly used as specifications for concurrent objects in the work on concurrent algorithms (see, e.g., [11]).[2]

4.3. Semantics of programs

We move on to the second stage of our semantics, which defines the evaluation of traces. Suppose we are given a trace τ and an initial state s, which is a function from variables $x, y, z, \ldots$ to integers.[5] The second stage is the evaluation of the

[5] All the results of the paper except the completeness can be developed without assuming any specific form of s. Here we do not take this general approach, to avoid being too abstract.

trace τ with s, and it is formally described by the evaluation function eval below[6]:

$$\begin{array}{rcl}
\mathsf{eval} & : & \mathit{States} \times \mathit{WTraces} \to \mathcal{P}(\mathit{States}) \\
\mathsf{eval}(s, \tau; (t, \mathit{call}\, o.f(n))) & = & \mathsf{eval}(s, \tau) \\
\mathsf{eval}(s, \tau; (t, \mathit{ret}(n)\, o.f)) & = & \mathsf{eval}(s, \tau) \\
\mathsf{eval}(s, \tau; (t, a)) & = & \bigcup_{s' \in \mathsf{eval}(s,\tau)} \{s'' \mid (s', s'') \in [\![a]\!]\} \\
\mathsf{eval}(s, \epsilon) & = & \{s\}.
\end{array}$$

The semantic clause for atomic client operations (t, a) assumes that we already have an interpretation $[\![a]\!]$ where a means a binary relation on *States*. Note that a state s does not change during method calls and returns. This is because firstly, in the evaluation map, a state describes the values of client variables only, not the internal status of objects and secondly, the assignment of a return value n to a variable x in $x{:=}o.f(e)$ is handled by a separate client operation; see the definition of $T(x{:=}o.f(e))$ in Fig. 5.

Now we combine the two stages, and give the semantics of programs P. Given a specific object system OS, the formal semantics $[\![P]\!](OS)$ is defined as follows:

$$\begin{array}{rcl}
[\![P]\!](OS) & : & \mathit{States} \to \mathcal{P}(\mathit{States}) \\
[\![P]\!](OS)(s) & = & \bigcup\{\, \mathsf{eval}(s, \tau) \mid \tau \in T(P) \wedge \mathsf{getHistory}(\tau) \in OS\,\}.
\end{array}$$

The semantics first calculates all traces $T(P)$ for τ, and then selects only those traces whose interactions with objects can be implemented by OS. Finally, the semantics runs all the selected traces with the initial state s.

4.4. *Observational refinement*

Our semantics observes the initial and final values of variables in threads, and ignores the object histories. We use this notion of observation and compare two different object systems OS_A and OS_C.

Definition 22. Let OS_A and OS_C be object systems. We say that

- OS_C **observationally refines** $OS_A \iff \forall P, s.\ [\![P]\!](OS_C)(s) \subseteq [\![P]\!](OS_A)(s)$;
- OS_C is **observationally equivalent** to $OS_A \iff \forall P.\ [\![P]\!](OS_C) = [\![P]\!](OS_A)$.

Usually, OS_A is a sequential local object system that serves as a specification, and OS_C is a concurrent object system representing the implementation. Observational refinement means that we can replace OS_A by OS_C in any programs without introducing new externally observable behaviours of those programs, and gives a sense that OS_C is a correct implementation of OS_A. We note that to obtain our results, we do not need to assume that the specification is either a sequential object system or a local one.[2]

In the next section, we will focus on answering the question: how do correctness conditions on concurrent objects, such as linearizability, relate to observational refinement?

5. The relationship to observational refinement

In this section we give an answer to the question posed at the end of Section 4, by relating sequential consistency and linearizability with observational refinement. In Section 6, we use abstraction in an attempt to explain the reason that such a relation exists.

This section is comprised of two parts. In the first part, Section 5.1, we describe a general method for proving observational refinement. In the second part, Section 5.2, we show that both linearizability and sequential consistency can be understood as specific instances of this method. For expository reasons, we first develop our results for object systems comprised only of histories in which every *call* action has a matching *return* and address the general case in Section 5.3.

Definition 23 (*Pending Operations*)**.** Thread t has a **pending operation** in a well-formed history H if the last action made by t in H is a *call* object action.

Definition 24 (*Quiescent Histories and Object Systems*)**.** A well-formed history H is **quiescent** if no thread has a pending action in H. An object system OS is **quiescent** if every history $H \in OS$ is quiescent. Given an object system OS, we denote by $\mathsf{getQuiescent}(OS)$ the maximal quiescent subset of OS.

Example 25. All the histories shown in Fig. 1 are quiescent. All the odd-length prefixes of histories H_0, H_1, and H_2 and all the non-empty prefixes of history H_3 are not quiescent. □

Note that, as intended, every *call* action in a quiescent history has a matching *return*. Also notice that the trace model, defined in Fig. 5, produces only traces whose projection on object actions yields quiescent histories.

6 $\mathcal{P}(\mathit{States})$ denotes the power set of *States*.

5.1. Simulation relations on histories

Roughly speaking, our general method for proving observational refinement works as follows. Suppose that we want to prove that OS_C observationally refines OS_A. We first need to choose a binary relation $\mathcal{R}$ on histories. This relation has to be a *simulation*, i.e., a relation that satisfies a specific requirement, which we will describe shortly. Next, we should prove that every history H in OS_C is $\mathcal{R}$-related to some history H' in OS_A. Once we finish both steps, the soundness theorem of our method lets us infer that OS_C is an observational refinement of OS_A.

The key part of the method, of course, lies in the requirement that the chosen binary relation $\mathcal{R}$ be a simulation. If we were allowed to use any relation for $\mathcal{R}$, we could pick the relation that relates all pairs of histories, and this would lead to the incorrect conclusion that every OS_C observationally refines OS_A, as long as OS_A is non-empty.

To describe our requirement on $\mathcal{R}$ and its consequences precisely, we need to formalize dependency between actions in a single trace, and define trace equivalence based on this formalization.

Definition 26 (*Independent Actions*)**.** An action φ **is independent of** an action φ', denoted as $\varphi\#\varphi'$, iff (i) $\mathsf{getTid}(\varphi) \neq \mathsf{getTid}(\varphi')$ and (ii) $\mathsf{eval}(s, \varphi\varphi') = \mathsf{eval}(s, \varphi'\varphi)$ for all $s \in \mathit{States}$.

Definition 27 (*Dependency Relations*)**.** For each trace τ, we define the **immediate dependency relation** $<_\tau$ to be the following relation on actions in τ[7]:

$$\tau_i <_\tau \tau_j \iff i < j \wedge \neg(\tau_i \# \tau_j).$$

The **dependency relation** $<_\tau^+$ on τ is the transitive closure of $<_\tau$.

Definition 28 (*Trace Equivalence*)**.** Traces τ, τ' are **equivalent**, denoted as $\tau \sim \tau'$, iff there exists a bijection $\pi : \{1, \ldots, |\tau|\} \to \{1, \ldots, |\tau'|\}$ such that

$$(\forall i.\ \tau_i = \tau'_{\pi(i)}) \wedge (\forall i, j.\ \tau_i <_\tau^+ \tau_j \iff \tau'_{\pi(i)} <_{\tau'}^+ \tau'_{\pi(j)}).$$

Intuitively, our notion of independence of actions is based on commutativity. (In particular, by the definition of #, an *object* action ψ can depend on another action ψ' only when both actions are done by the same thread.) Thus, informally, $\tau \sim \tau'$ means that τ' can be obtained by swapping independent actions in τ. Since we swap only independent actions, we expect τ' and τ to essentially mean the same computation. The lemma below justifies this expectation, by showing that our semantics cannot observe the difference between equivalent traces.

Lemma 29. *For all* $\tau, \tau' \in \mathit{WTraces}$, *if* $\tau \sim \tau'$, *then* $(\forall P.\ \tau \in T(P) \iff \tau' \in T(P))$ *and* $(\forall s.\ \mathsf{eval}(s, \tau) = \mathsf{eval}(s, \tau'))$.

We are now ready to give the definition of simulation, which encapsulates our requirement on relations on histories, and to prove the soundness of our proof method based on simulation.

Definition 30 (*Simulation*)**.** A binary relation $\mathcal{R}$ on well-formed histories is a **simulation** iff for all well-formed histories H and H' such that $(H, H') \in \mathcal{R}$,

$$\begin{aligned}\forall \tau \in \mathit{WTraces}.\ & \mathsf{getHistory}(\tau) = H \\ & \Longrightarrow \exists \tau' \in \mathit{WTraces}.\ \tau \sim \tau' \wedge \mathsf{getHistory}(\tau') = H'.\end{aligned}$$

One way to understand this definition is to consider the function means defined by

$$\begin{aligned}\mathsf{means} \quad &: \quad \mathit{WHist} \to \mathcal{P}(\mathit{WTraces}) \\ \mathsf{means}(H) \quad &= \quad \{\tau \in \mathit{WTraces} \mid \mathsf{getHistory}(\tau) = H\},\end{aligned}$$

and to read a history H as a representation of the trace set $\mathsf{means}(H)$. Intuitively, the trace set $\mathsf{means}(H)$ consists of the well-formed traces whose interactions with objects are precisely H. According to this reading, the requirement in the definition of simulation simply means that $\mathsf{means}(H)$ is a subset of $\mathsf{means}(H')$ modulo trace equivalence $\sim$. For every relation $\mathcal{R}$ on histories, we now define its lifting to a relation $\lhd_{\mathcal{R}}$ on object systems.[8]

Definition 31 (*Lifted Relations*)**.** Let $\mathcal{R}$ be a binary relation on well-formed histories. The lifting of $\mathcal{R}$ to *quiescent* object systems OS_A and OS_C, denoted as $OS_C \lhd_{\mathcal{R}} OS_A$, is

$$OS_C \lhd_{\mathcal{R}} OS_A \iff \forall H \in OS_C.\ \exists H' \in OS_A.\ (H, H') \in \mathcal{R}.$$

Theorem 32 (*Simulation*)**.** *If* $OS_C \lhd_{\mathcal{R}} OS_A$ *and* $\mathcal{R}$ *is a simulation, the object system* OS_C *observationally refines* OS_A.

[7] Strictly speaking, $<_\tau$ is a relation on the indices $\{1, \ldots, |\tau|\}$ of τ so we should have written $i <_\tau j$. In this paper, we use a rather informal notation $\tau_i <_\tau \tau_j$ instead, since we found this notation easier to understand.

[8] We remind the reader that in this section we assume that both OS_A and OS_C are *quiescent* object systems.

Proof. Consider a program P and states s, s' such that $s' \in [\![P]\!](OS_C)(s)$. Then, by the definition of $[\![P]\!]$, there exist a well-formed trace $\tau \in T(P)$ and a history $H \in OS_C$ such that $\mathsf{getHistory}(\tau) = H$ and $s' \in \mathsf{eval}(s, \tau)$. Since $H \in OS_C$ and $OS_C \lhd_{\mathcal{R}} OS_A$ by our assumption, there exists $H' \in OS_A$ with $(H, H') \in \mathcal{R}$. Furthermore, H and H' are well-formed, because object systems contain only well-formed histories. Now, since $\mathcal{R}$ is a simulation, τ is well-formed and $\mathsf{getHistory}(\tau) = H$, there exists a well-formed trace τ' such that

$$\tau \sim \tau' \wedge \mathsf{getHistory}(\tau') = H'.$$

Note that because of Lemma 29, the first conjunct here implies that $\tau' \in T(P)$ and $s' \in \mathsf{eval}(s, \tau')$. This and the second conjunct $\mathsf{getHistory}(\tau') = H'$ together imply $s' \in [\![P]\!](OS_A)(s)$ as desired. □

5.2. Sequential consistency, linearizability and observational refinement

Now we explain the first two main results of this paper: (i) linearizability coincides with observational refinement if client programs are allowed to use at least one global variable; (ii) sequential consistency coincides with observational refinement if client operations of different threads are independent and so commutative.

It is not difficult to obtain a high-level understanding of why linearizability implies observational refinement and why sequential consistency does the same under some condition on client operations. Both linearizability and sequential consistency define certain relationships between two object systems, one of which is normally assumed sequential and local. Interestingly, in both cases, we can prove that these relationships are generated by lifting some *simulation* relations. From this observation follows our soundness results, because Theorem 32 says that all such simulation-generated relationships on object systems imply observational refinements.

In the rest of this section, we will spell out the details of the high-level proof sketches just given. For this, we need to review the relations on histories used by sequential consistency and linearizability [12].

Definition 33 (*Weakly Equivalent Histories*)**.** Two histories are **weakly equivalent**, denoted as $H \equiv H'$, iff their projections to threads are equal[9]:

$$H \equiv H' \iff \forall t.\ H|_t = H'|_t.$$

As its name indicates, the weak equivalence is indeed a weak notion. It only says that the two traces are both interleavings of the same sequential threads (but they could be different interleavings).

Definition 34 (*Happens-Before Order*)**.** For a well-formed history H, the **happens-before order** $\prec_H$ is a binary relation on object actions in H defined by

$$\begin{aligned} H_i \prec_H H_j \iff \exists i', j'.\ & i \le i' < j' \le j \wedge \\ & \mathsf{retAct}(H_{i'}) \wedge \mathsf{callAct}(H_{j'}) \wedge \\ & \mathsf{getTid}(H_i) = \mathsf{getTid}(H_{i'}) \wedge \mathsf{getTid}(H_{j'}) = \mathsf{getTid}(H_j). \end{aligned}$$

Here $\mathsf{retAct}(\psi)$ holds when ψ is a return and $\mathsf{callAct}(\psi)$ holds when ψ is a call.

This definition is intended to express that in the history H, the method call for H_i is completed before the call for H_j starts. To see this intention, note that H is well-formed. One important consequence of this well-formedness is that if an object action ψ of some thread t is followed by some return action ψ' of the same thread in the history H (i.e., $H = \ldots\psi\ldots\psi'\ldots$), then the return for ψ itself appears before ψ' or it is ψ'. Thus, the existence of $H_{i'}$ in the definition ensures that the return action for H_i appears before or at $H_{i'}$ in the history H. By a similar argument, we can see that the call for H_j appears after or at $H_{j'}$. Since $i' < j'$, these two observations mean that the return for H_i appears before the call for H_j, which is the intended meaning of the definition. Using this happens-before order, we define the linearizability relation $\sqsubseteq$:

Definition 35 (*Linearizability Relation*)**.** The linearizability relation is a binary relation $\sqsubseteq$ on histories defined as follows: $H \sqsubseteq H'$ iff (i) $H \equiv H'$ and (ii) there is a bijection $\pi : \{1, \ldots, |H|\} \to \{1, \ldots, |H'|\}$ such that

$$(\forall i.\ H_i = H'_{\pi(i)}) \wedge (\forall i, j.\ H_i \prec_H H_j \implies H'_{\pi(i)} \prec_{H'} H'_{\pi(j)}).$$

Recall that for each relation $\mathcal{R}$ on histories, its lifting $\lhd_{\mathcal{R}}$ to the relation on object systems is defined by: $OS \lhd_{\mathcal{R}} OS' \iff \forall H{\in}OS.\ \exists H'{\in}OS'.\ (H, H') \in \mathcal{R}$. Using this lifting, we formally specify sequential consistency and linearizability for *quiescent* object systems.[10]

Definition 36. Let OS_A and OS_C be *quiescent* object systems. We say that OS_C is **sequentially consistent** with respect to OS_A iff $OS_C \lhd_{\equiv} OS_A$.

[9] For the same definition, Herlihy and Wing [12] use the terminology "equivalence".

[10] We remind the reader that in this section we assume that both OS_A and OS_C are *quiescent* object systems (Definition 24). We note, however, that both linearizability and sequential consistency are well defined as relations on arbitrary well-formed histories.

Definition 37. Let OS_A and OS_C be *quiescent* object systems. We say that OS_C is **linearizable** with respect to OS_A iff $OS_C \lhd_{\sqsubseteq} OS_A$.

Note that this definition does not assume the sequentiality and locality of OS_A, unlike Herlihy and Wing's definitions. We use this more general definition here in order to emphasize that the core ideas in the technical notions of sequential consistency and linearizability lie in relations $\equiv$ and $\sqsubseteq$ on histories, not in the use of a sequential local object system (as a specification).

5.2.1. Soundness

In this section, we show that linearizability implies observational refinement, and that if client operations of different threads are independent, and thus commute, sequential consistency implies observational refinement.

We first prove the theorem that connects linearizability and observational refinement. Our proof uses the lemma below:

Lemma 38. *Let H be a well-formed history and let i, j be indices in* $\{1, \ldots, |H|\}$. *Then,*

$$(\exists \tau \in \mathit{WTraces}.\ \mathsf{getHistory}(\tau) = H \wedge H_i <_\tau^+ H_j)$$
$$\implies (i < j) \wedge (\mathsf{getTid}(H_i) = \mathsf{getTid}(H_j) \vee H_i \prec_H H_j).$$

Proof. Consider a well-formed history H, indices i, j of H and a well-formed trace τ such that the assumptions of this lemma hold. Then, we have indices $i_1 < i_2 < \cdots < i_n$ of τ such that

$$H_i = \tau_{i_1} \quad <_\tau \quad \tau_{i_2} \quad <_\tau \quad \cdots \quad <_\tau \quad \tau_{i_{n-1}} \quad <_\tau \quad \tau_{i_n} = H_j. \tag{1}$$

One conclusion $i < j$ of this lemma follows from this, because $\mathsf{getHistory}(\tau) = H$ means that the order of object actions in H are maintained in τ. To obtain the other conclusion of the lemma, let $t = \mathsf{getTid}(H_i)$ and $t' = \mathsf{getTid}(H_j)$. Suppose that $t \neq t'$. We will prove that for some $i_k, i_l \in \{i_1, \ldots, i_n\}$,

$$\begin{aligned} t = \mathsf{getTid}(H_i) = \mathsf{getTid}(\tau_{i_k}) \wedge \mathsf{retAct}(\tau_{i_k}) \wedge \\ i_k < i_l \wedge \\ t' = \mathsf{getTid}(H_j) = \mathsf{getTid}(\tau_{i_l}) \wedge \mathsf{callAct}(\tau_{i_l}). \end{aligned} \tag{2}$$

Note that this gives the conclusion that we are looking for, because all object actions in τ are from H and their relative positions in τ are the same as those in H. In the rest of the proof, we focus on showing (2) for some i_k, i_l. Recall that by the definition of #, an object action ψ can depend on another action φ only when both actions are done by the same thread. Now note that the first and last actions in the chain in (1) are *object* actions by *different* threads t and t'. Thus, the chain in (1) must contain *client* operations τ_{i_x} and τ_{i_y} such that $\mathsf{getTid}(\tau_{i_x}) = t$ and $\mathsf{getTid}(\tau_{i_y}) = t'$. Let τ_{i_a} be the first client operation by the thread t in the chain and let τ_{i_b} be the last client operation by t'. Then, $i_a < i_b$. This is because otherwise, the sequence $\tau_{i_a}\ \tau_{i_a+1}\ \ldots\ \tau_{i_n}$ does not have any client operation of the thread t', while τ_{i_a} is an action of the thread t and τ_{i_n} is an action of the different thread t'; these facts make it impossible to have $\tau_{i_a} <_\tau \tau_{i_a+1} <_\tau \cdots <_\tau \tau_{i_n}$.

τ_{i_1} is an object action by the thread t and τ_{i_a} is a client operation by the same thread. Thus, by the well-formedness of τ, there should exist some i_k between i_1 (including) and i_a such that τ_{i_k} is a return object action by the thread t. By a symmetric argument, there should be some i_l between i_b and i_n (including) such that τ_{i_l} is a call object action by t'. We have just shown that i_k and i_l satisfy (2), as desired. □

Theorem 39. *The linearizability relation* $\sqsubseteq$ *is a simulation.*

Proof. For an action φ and a trace τ, define $\varphi\#\tau$ to mean that $\varphi\#\tau_j$ for all $j \in \{1, \ldots, |\tau|\}$. In this proof, we will use this $\varphi\#\tau$ predicate and the following facts:

Fact 1. Trace equivalence $\sim$ is symmetric and transitive.
Fact 2. If $\tau \sim \tau'$ and τ is well-formed, τ' is also well-formed.
Fact 3. If $\tau\tau'$ is well-formed, its prefix τ is also well-formed.
Fact 4. If $\varphi\#\tau'$, we have that $\tau\varphi\tau' \sim \tau\tau'\varphi$.
Fact 5. If $\tau \sim \tau'$, we have that $\tau\varphi \sim \tau'\varphi$.

Consider well-formed histories H, S and a well-formed trace τ such that $H \sqsubseteq S$ and $\mathsf{getHistory}(\tau) = H$. We will prove the existence of a trace σ such that $\tau \sim \sigma$ and $\mathsf{getHistory}(\sigma) = S$. This gives the desired conclusion of this theorem; the only missing requirement for proving that $\sqsubseteq$ is a simulation is the well-formedness of σ, but it can be inferred from $\tau \sim \sigma$ and the well-formedness of τ by Fact 2.

Our proof is by induction on the length of S. If $|S| = 0$, H has to be the empty sequence as well. Thus, we can choose τ as the required σ in this case. Now suppose that $|S| \neq 0$. That is, $S = S'\psi$ for some history S' and object action ψ. Note that since the well-formed traces are closed under prefix (Fact 3), S' is also a well-formed history. During the proof, we will use this fact, especially when applying induction on S'.

Let δ be the projection of τ to client operations (i.e., $\delta = \mathsf{getClient}(\tau)$). The starting point of our proof is to split τ, H, δ. By assumption, $H \sqsubseteq S'\psi$. By the definition of $\sqsubseteq$, this means that

$$\begin{aligned}\exists H', H''.\ \ & H = H'\psi H'' \wedge H'H'' \sqsubseteq S' \\ & \wedge \big(\forall j \in \{1, \dots, |H''|\}.\ \neg(\psi \prec_H H''_j) \wedge \mathsf{getTid}(\psi) \neq \mathsf{getTid}(H''_j)\big).\end{aligned} \tag{3}$$

Here we use the bijection between indices of H and $S'\psi$, which exists by the definition of $H \sqsubseteq S'\psi$. The action ψ in $H'\psi H''$ is what is mapped to the last action in $S'\psi$ by this bijection. The last conjunct of (3) says that the thread-id of every action of H'' is different from $\mathsf{getTid}(\psi)$. Thus, $\psi \# H''$ (because an *object* action is independent of all actions by *different* threads). From this independence and the well-formedness of H, we can derive that $H'H''\psi$ is well-formed (Facts 2 and 4), and that its prefix $H'H''$ is also well-formed (Fact 3). Another important consequence of (3) is that since $\tau \in \mathit{interleave}(\delta, H)$, the splitting $H'\psi H''$ of H induces splittings of τ and δ as follows: there exist $\tau', \tau'', \delta', \delta''$ such that

$$\tau = \tau'\psi\tau'' \wedge \delta = \delta'\delta'' \wedge \tau' \in \mathit{interleave}(\delta', H') \wedge \tau'' \in \mathit{interleave}(\delta'', H''). \tag{4}$$

The next step of our proof is to identify one short-cut for showing this theorem. The short-cut is to prove $\psi \# \tau''$. To see why this short-cut is sound, suppose that $\psi \# \tau''$. Then, by Fact 4,

$$\tau = \tau'\psi\tau'' \sim \tau'\tau''\psi. \tag{5}$$

Since τ is well-formed, this implies that $\tau'\tau''\psi$ and its prefix $\tau'\tau''$ are well-formed traces as well (Facts 2 and 3). Furthermore, $\mathsf{getHistory}(\tau'\tau'') = H'H''$, because of the last two conjuncts of (4). Thus, we can apply the induction hypothesis to $\tau'\tau'', H'H'', S'$, and obtain σ with the property: $\tau'\tau'' \sim \sigma \wedge \mathsf{getHistory}(\sigma) = S'$. From this and Fact 5, it follows that

$$\tau'\tau''\psi \sim \sigma\psi \wedge \mathsf{getHistory}(\sigma\psi) = \mathsf{getHistory}(\sigma)\psi = S'\psi. \tag{6}$$

Now, the formulas (5) and (6) and the transitivity of $\sim$ (Fact 1) imply that $\sigma\psi$ is the required trace by this theorem. In the remainder of the proof, we will use this short-cut, without explicitly mentioning it.

The final step is to do the case analysis on δ''. Specifically, we use the nested induction on the length of δ''. Suppose that $|\delta''| = 0$. Then, $\tau'' = H''$, and by the last conjunct of (3), i.e., the universal formula, we have that $\psi \# \tau''$; since ψ is an object action, it is independent of actions by different threads. The theorem follows from this. Now consider the inductive case of this nested induction: $|\delta''| > 0$. Note that if $\psi \# \delta''$, then $\psi \# \tau''$, which implies the theorem. So, we are going to assume that $\neg(\psi \# \delta'')$. Pick the greatest index i of τ'' such that $\psi <^+_\tau \tau''_i$. Let $\varphi = \tau''_i$. Because of the last conjunct of (3) and Lemma 38, τ''_i comes from δ, not H''. In particular, this ensures that there are following further splittings of δ'', τ'' and H'': for some traces $\gamma, \gamma', \kappa, \kappa', T, T'$,

$$\begin{aligned}& \delta'' = \gamma\varphi\gamma' \wedge \tau'' = \kappa\varphi\kappa' \wedge H'' = TT' \wedge \\ & \kappa \in \mathit{interleave}(\gamma, T) \wedge \kappa' \in \mathit{interleave}(\gamma', T') \wedge \varphi \# \kappa'.\end{aligned}$$

Here the last conjunct $\varphi \# \kappa'$ comes from the fact that φ is the last element of τ'' with $\psi <^+_\tau \varphi$. Since γ' is a subsequence of κ', the last conjunct $\varphi \# \kappa'$ implies that $\varphi \# \gamma'$. Also, $\tau'\psi\kappa\varphi\kappa' \sim \tau'\psi\kappa\kappa'\varphi$ by Fact 4. Now, since $\tau = \tau'\psi\kappa\varphi\kappa'$ is well-formed, the equivalent trace $\tau'\psi\kappa\kappa'\varphi$ and its prefix $\tau'\psi\kappa\kappa'$ both are well-formed as well (Facts 2 and 3). Furthermore, $\tau'\psi\kappa\kappa' \in \mathit{interleave}(\delta'\gamma\gamma', H'\psi H'')$. Since the length of $\gamma\gamma'$ is shorter than δ'', we can apply the induction hypothesis of the nested induction, and get

$$\exists\sigma.\quad \tau'\psi\kappa\kappa' \sim \sigma \wedge \mathsf{getHistory}(\sigma) = S'\psi. \tag{7}$$

We will prove that $\sigma\varphi$ is the trace desired for this theorem. Because of $\varphi \# \kappa'$ and Fact 4, $\tau = \tau'\psi\kappa\varphi\kappa' \sim \tau'\psi\kappa\kappa'\varphi$. Also, because of Fact 5 and the first conjunct of (7), $\tau'\psi\kappa\kappa'\varphi \sim \sigma\varphi$. Thus, $\tau \sim \sigma\varphi$ by the transitivity of $\sim$. Furthermore, since φ is not an object action, the second conjunct of (7) implies that $\mathsf{getHistory}(\sigma\varphi) = \mathsf{getHistory}(\sigma) = S'\psi$. We have just shown that $\sigma\varphi$ is the desired trace. □

Corollary 40. *If OS_C is linearizable with respect to OS_A, then OS_C observationally refines OS_A.*

Next, we consider sequential consistency. For sequential consistency to imply observational refinement, we restrict programs such that client operations of different threads are independent:

$$\forall t, t', a, a'.\ (t \neq t' \wedge a \in \mathit{Cop}_t \wedge a' \in \mathit{Cop}_{t'}) \implies a \# a'.$$

Lemma 41. *Suppose that client operations of different threads are independent. Then, for all well-formed histories H and indices i, j in $\{1, \dots, |H|\}$,*

$$\begin{aligned}& \big(\exists\tau \in \mathit{WTraces}.\ \mathsf{getHistory}(\tau) = H \wedge H_i <^+_\tau H_j\big) \\ & \qquad \implies \big(i < j \wedge \mathsf{getTid}(H_i) = \mathsf{getTid}(H_j)\big).\end{aligned}$$

Proof. Consider a well-formed history H, indices i, j and a well-formed trace τ satisfying the assumptions of this lemma. Then, for some indices $i_1 < \cdots < i_n$ of τ,

$$H_i = \tau_{i_1} \quad <_\tau \quad \tau_{i_2} \quad <_\tau \quad \cdots \quad <_\tau \quad \tau_{i_{n-1}} \quad <_\tau \quad \tau_{i_n} = H_j. \tag{8}$$

One conclusion $i < j$ of this lemma follows from this; the assumption $\mathsf{getHistory}(\tau) = H$ of this lemma means that the orders of object actions in H are maintained in τ. To obtain the other conclusion of the lemma, we point out one important property of #: under the assumption of this lemma, $\neg(\varphi \# \varphi')$ only when $\mathsf{getTid}(\varphi) = \mathsf{getTid}(\varphi')$. (Here φ, φ' are not necessarily object actions.) To see why this property holds, we assume $\neg(\varphi \# \varphi')$ and consider all possible cases of φ and φ'. If one of φ and φ' is an object action, the definition of # implies that φ and φ' have to be actions by the same thread. Otherwise, both φ and φ' are atomic client operations. By our assumption, two client operations are independent if they are performed by different threads. This implies that φ and φ' should be actions by the same thread. Now, note that $\tau_k <_\tau \tau_l$ implies $\neg(\tau_k \# \tau_l)$, which in turn entails $\mathsf{getTid}(\tau_k) = \mathsf{getTid}(\tau_l)$ by what we have just shown. Thus, we can derive the following desired equality from (8): $\mathsf{getTid}(H_i) = \mathsf{getTid}(\tau_{i_1}) = \mathsf{getTid}(\tau_{i_2}) = \cdots = \mathsf{getTid}(\tau_{i_n}) = \mathsf{getTid}(H_j)$. □

Theorem 42. *If client operations of different threads are independent, the weak equivalence $\equiv$ is a simulation.*

Proof. The proof is similar to the one for Theorem 39. Instead of repeating the common parts of these two proofs, we will explain what we need to change in the proof of Theorem 39, so as to obtain the proof of this theorem. Firstly, we should replace linearizability relation $\sqsubseteq$ by weak equivalence $\equiv$. Secondly, we need to change the formula (3) to

$$\exists H'H''.\ H = H'\psi H'' \wedge H'H'' \equiv S' \wedge \forall j \in \{1, \ldots, |H''|\}.\ \mathsf{getTid}(\psi) \neq \mathsf{getTid}(H''_j).$$

Finally, we should use Lemma 41 instead of Lemma 38. After these three changes have been made, the result becomes the proof of this theorem. □

Corollary 43. *If OS_C is sequentially consistent with respect to OS_A and client operations of different threads are independent, OS_C is an observational refinement of OS_A.*

5.2.2. Completeness

In this section, we show that under suitable assumptions on programming languages and object systems, we can obtain the converse of Corollaries 40 and 43: observational refinement implies linearizability and sequential consistency. Firstly, we remind the reader that we assume that the object systems are quiescent (Definition 24). This assumption is necessary at this point, because observational refinement considers only terminating, completed computations. In particular, every call action should have a matching return action. (In Section 5.3 we add the notion of *completability*, which allows us to handle non-quiescent object systems.) Secondly, we assume that threads' primitive commands include the `skip` statement. Finally, we consider specific assumptions for sequential consistency and linearizability, which will be described shortly.

Sequential consistency. For sequential consistency, we suppose that the programming language contains atomic assignments $x{:=}n$ of constants n to thread-local variable x and has atomic assume statements of the form $\mathtt{assume}(x{=}n)$ with thread-local variable x.[11] Under this supposition, observational refinement implies sequential consistency. (Note that this supposition is consistent with the assumption of Corollary 43: Client operations of different threads are independent. In particular, note that the use of global variables, or any other kinds of thread-shared resources which allow for inter-thread communication that can bypass the object system, is not allowed.)

Theorem 44. *If OS_C observationally refines OS_A then $OS_C \lhd_\equiv OS_A$.*

The main idea of the proof is to create for every history $H \in OS_C$ a program P_H which records the interaction of every thread t with the object system using t's local variables. The detailed proof is given below.

Proof. The plan of the proof is to construct for every history $H \in OS_C$ a program P_H that records the interactions of every thread with the object system from the point of view of every thread. The goal is to make it possible to record this interaction at every final state s' of P_H and thus be able to read $H|_t$ from s'.

Let H be a history in OS_C. Let $H|_t$ be the projection of H on the object actions executed by thread t. H, when viewed as a trace, is well-formed. Thus, $H|_t$, is a sequential trace. In particular, $H|_t$ is comprised of a sequence of pairs of call and return object actions. (By our assumption, $H|_t$ is comprised only of matching pairs of calls and returns.)

For every thread t that has an action in H, we construct a straight-line command $P^t_H = C^t_1; C^t_2; \ldots; C^t_{k_t}$, where $k_t = |(H|_t)|/2$ is the number of pairs of call and return actions executed by thread t. Here C^t_i is a sequence of atomic commands, and it is constructed according to the i-th pair of object actions in $H|_t$. The definition of C^t_i goes as follows. For $i = 1, \ldots, k$, let $(H|_t)_{2i-1} = (t, \mathit{call}\, o.f(n^t_i))$ and $(H|_t)_{2i} = (t, \mathit{ret}(m^t_i)\, o.f)$. The composed command that we construct for this pair is

$$C^t_i = x^t_i{:=}n^t_i;\ \ y^t_i{:=}o.f(x^t_i);\ \ \mathtt{assume}(y^t_i{=}m^t_i).$$

[11] Technically, this assumption also means that $T(x{:=}n)t$ and $T(\mathtt{assume}(x{=}n))t$ are singleton traces (t, a) and (t, b), respectively, where $[\![a]\!](s){\equiv}\{s[x{\mapsto}n]\}$ and $[\![b]\!](s){\equiv}\text{if } (s(x){=}n) \text{ then } \{s\} \text{ else } \{\}$.

Note that the composed command is constructed according to the values in H: C_i^t invokes operation f on object o passing n_i^t as the argument and expects the return value to be m_i^t. Furthermore, note that the argument to the command is recorded in x_i^t and the return value is recorded in y_i^t. Both of these variables are local to thread t and are never rewritten. Thus, starting from any state, the only trace $\tau_i^t \in T(C_i^t)t$ that can be executed until completion for some initial state is

$$\begin{aligned}\tau^t_{n_i^t,m_i^t} &= (t, x_i^t := n_i^t); (t, \mathtt{assume}(x_i^t = n_i^t));\\ &\quad (t, \mathit{call}\, o.f(n_i^t)); (t, \mathit{ret}(m_i^t));\\ &\quad (t, y_i^t := m_i^t); (t, \mathtt{assume}(y_i^t = m_i^t)).\end{aligned}$$

Here we overload $x := n$ and $\mathtt{assume}(x{=}n)$ to mean not primitive commands but atomic client operations in Cop_t, with standard meanings. Let τ^t be $\tau^t_{n_1^t,m_1^t}; \tau^t_{n_2^t,m_2^t}; \ldots$.

The following claims are immediate:

Claim 1. For every trace τ, if $\tau|_t \in T(P_H^t)t$ and τ can be evaluated until completion for some initial state, then $\tau|_t = \tau^t$. Furthermore, if $\tau|_t = \tau^t$ for all threads t in τ, the evaluation of τ can be evaluated until completion for all states.
Claim 2. Since x_i^t and y_i^t are thread-local variables and they are never rewritten by their owner thread, we have that $x_i^t = n_i^t$ and $y_i^t = m_i^t$ at every final state in $\mathsf{eval}(s, \tau)$, as long as $\tau|_t \in T(P_H^t)t$.
Claim 3. $H|_t = \mathsf{getHistory}(\tau^t)$.

We construct a program P_H which corresponds to history H by a parallel composition of the commands for every thread: $P_H = P_H^1 || \ldots || P_H^{t_{max}}$, where t_{max} is the maximal thread identifier in H. For technical reasons, if $1 \le t < t_{max}$ does not appear in H, we define $P_H^t = \mathtt{skip}$. (Recall that H is a finite sequence; thus there is a finite number of threads executing in H. Specifically, there are finitely many such commands).

Let τ_H be an interleaving of $\tau^{t_1}, \ldots, \tau^{t_{max}}$ such that $\mathsf{getHistory}(\tau_H) = H$. Such a τ_H exists because of Claim 3 above and $H \in \mathit{interleave}(H|_{t_1}, \ldots, H|_{t_{max}})$. Furthermore, $\tau_H \in T(P_H)$ because $\tau^{t_i} \in T(P_H^{t_i})t_i$ for every $t_i = t_1, \ldots, t_{max}$.

Let n_0 be a value that does not appear in H. Let s_0 be a state where all (local) variables x_i^t's and y_i^t's are initialized to n_0. Let s' be the state where $s'(x_i^t) = n_i^t$ and $s'(y_i^t) = m_i^t$. Because $\mathsf{getHistory}(\tau_H) = H \in OS_C$, the combination of Claims 1 and 2 and the definition of eval implies that $s' \in [\![P_H]\!](OS_C)(s_0)$.

Now, we have $s' \in [\![P_H]\!](OS_A)(s_0)$, because OS_C observationally refines OS_A. Thus, there exists a trace $\tau_A \in T(P_H)$ and history $H_A \in OS_A$ such that $s' \in \mathsf{eval}(\tau_A, s_0)$ and $\mathsf{getHistory}(\tau_A) = H_A$. Since $\tau_A \in T(P_H)$, we have that $(\tau_A)|_t \in T(P_H^t)$. By Claim 1, this means that for every thread t, $\tau_A|_t = \tau^t$. By Claim 3, we get that $H_A \equiv H$, which implies that OS_C is sequentially consistent with respect to OS_A. □

Linearizability. For linearizability, we further suppose that there is a single global variable g shared by all threads. That is, threads can assign constants to g atomically, or they can run the statement $\mathtt{assume}(g{=}n)$ for some constant n. Under this supposition, observational refinement implies linearizability.

Theorem 45. *If OS_C observationally refines OS_A, then $OS_C \lhd_{\sqsubseteq} OS_A$.*

The core idea of the proof is, again, to create for every history $H \in OS_C$ one specific program P_H. This program uses a single global variable and satisfies that for every (terminating) execution τ of P_H, the object history of τ always has the same happens-before relation as H. The details are described in the following proof.

Proof. The plan of the proof is similar to that for Theorem 44. We construct, for every history $H \in OS_C$, a program P_H that records the interactions of every thread with the object system. This recording remains in the final state of P_H and thus allows us to see every step of $H|_t$. We use a global variable g to enforce that every (terminating) execution of the program has the same happens-before order between object operations.

Let H be a history in OS_C. Let $H|_t$ be the projection of H to object operations executed by thread t. Using the same argument as in the proof of Theorem 44, we construct, for every thread t which has an action in H, a straight-line composite command

$$PL_H^t = CL_1^t;\ CL_2^t;\ \ldots;\ CL_{k_t}^t,$$

where $k_t = |(H|_t)|/2$ is the number of pairs of call and return actions executed by thread t, as a sequence of composed commands and CL_i^t is constructed according to the i-th pair of object actions in H done by thread t. The construction of CL_i^t goes as follows. For $i = 1, \ldots, k_t$, let $(H|_t)_{2i-1} = (t, \mathit{call}\, o.f(n_i^t))$ and $(H|_t)_{2i} = (t, \mathit{ret}(m_i^t)\, o.f)$. Let i_c and i_r be the indices of these actions in H, i.e., $H_{i_c} = (H|_t)_{2i-1}$ and $H_{i_r} = (H|_t)_{2i}$. The corresponding command CL_i^t is

$$CL_i^t = \mathtt{assume}(g{=}i_c);\ g{:=}i_c{+}1;\ C_i^t;\ \mathtt{assume}(g{=}i_r);\ g{:=}i_r{+}1\ ,$$

where C_i^t is defined as in the proof of Theorem 44, i.e.,

$$C_i^t = x_i^t{:=}n_i^t;\ y_i^t{:=}o.f(x_i^t);\ \mathtt{assume}(y_i^t{=}m_i^t)\ .$$

Note that the command C_i^t in CL_i^t can be executed only when $g = i_c$, and that if it is, CL_i^t increments g by 1 before running C_i^t. Similarly, after C_i^t terminates, the computation of t can continue only when $g = i_r$, and then again g is incremented.

For the same reason as was discussed in the proof of Theorem 44, the only trace $\alpha_i^t \in T(CL_i^t)t$ that can be executed until completion is

$$\begin{aligned}\alpha^t_{n_i^t,m_i^t} = {} & (t, \mathtt{assume}(g{=}i_c));\ (t, g := i_c{+}1);\\ & \tau^t_{n_i^t,m_i^t};\\ & (t, \mathtt{assume}(g{=}i_r));\ (t, g := i_r{+}1)\end{aligned}$$

where $\tau^t_{n_i^t,m_i^t}$ is defined as in the proof of Theorem 44:

$$\begin{aligned}\tau^t_{n_i^t,m_i^t} = {} & (t, x_i^t{:=}n_i^t);\ (t, \mathtt{assume}(x_i^t{=}n_i^t));\\ & (t, call\ o.f(n_i^t));\ (t, ret(m_i^t));\\ & (t, y_i^t{:=}m_i^t);\ (t, \mathtt{assume}(y_i^t{=}m_i^t)).\end{aligned}$$

We construct a program P_H^L, which corresponds to history H, by a parallel composition of the command for each thread:

$$P_H^L = PL_H^1 || \dots || PL_H^{t_{max}},$$

where t_{max} is the maximal thread identifier in H. For technical reasons, in the case where some t with $1 \le t < t_{max}$ does not appear in H, we define $PL_H^t = \mathtt{skip}$.

Let α_H be an interleaving of $\alpha^1, \dots, \alpha^{t_{max}}$ such that $\mathsf{getHistory}(\alpha_H) = H$ that can be evaluated until completion. Again, such an α_H exists for the same reason as was discussed in the proof of Theorem 44. Furthermore, assume that α_H is an interleaving of action sequence fragments of the following two forms:

$$(t, \mathtt{assume}(g{=}i_c)); (t, g{:=}i_c{+}1); (t, x_i^t{:=}n_i^t); (t, \mathtt{assume}(x_i^t{=}n_i^t)); (t, call\ o.f(n_i^t)),$$

$$(t, ret(m_i^t)); (t, y_i^t{:=}m_i^t); (t, \mathtt{assume}(y_i^t{=}m_i^t)); (t, \mathtt{assume}(g{=}i_r)); (t, g{:=}i_r{+}1).$$

Thus, once a thread runs $\mathtt{assume}(g{=}i_c)$ in α_H, it continues without there being intervention by different threads at least until it invokes the i-th (call) object action. Similarly, once a thread runs the i-th (return) object action, it continues without being interrupted at least until it assigns $i_r{+}1$ to g. Note that in α, g is incremented and tested in the same order as the object actions occur in H.

Let n_0 be a value that does not appear in H. Let s_0 be a state where all (local) variables x_i^t's and y_i^t's are initialized to n_0 and g is initialized to 1. Note that our construction of α_H ensures that the trace α_H can run until completion from the initial state s_0. Furthermore, since OS_C observationally refines OS_A, by the same arguments as in the proof of Theorem 44, we can infer that there exists a history $S \in OS_A$ such that $S \equiv H$, and also that there exists $\alpha \in T(P_H^L)$ with $\mathsf{getHistory}(\alpha) = S \wedge \mathsf{eval}(s_0, \alpha) \neq \emptyset$. Using α, we will show that the bijection π implicit in $H \equiv S$ preserves the happens-before order. Let H_i be a return action in H and let H_j be a call action. Let $\bar{i}$ and $\bar{j}$ be the indices of H_i and H_j in S, respectively. By the choice of π, it is sufficient to prove that if $i < j$, then $\bar{i} < \bar{j}$. Suppose that $i < j$. Then, $i_r < j_c$. Since α can run until completion, the definition of P_H^L implies that the assume statement for g following the return of $S_{\bar{i}}$ should be run before the assume statement that comes before the call of $S_{\bar{j}}$. This means that $S_{\bar{i}}$ occurs before $S_{\bar{j}}$ in α. Since $\mathsf{getHistory}(\alpha) = S$, we should have that $\bar{i} < \bar{j}$ as desired. What we have shown implies that π preserves the happens-before order. This in turn means $H \sqsubseteq H_A^L$. That is, OS_C is linearizable with respect to OS_A. $\square$

5.3. The general case: Non-quiescent object systems

In Sections 5.1 and 5.2, we restricted our attention to quiescent object systems. We now remove this restriction. We first show that it is possible to establish the soundness result in the general case by, essentially, changing the definition of lifted relations (Definition 31). We then discuss different ways to achieve completeness in the general case.

5.3.1. Soundness

In Section 5.1, we proved that linearizability implies observational refinement in two stages. First, we showed that the linearizability relation over well-formed histories is a simulation relation (Theorem 39). Next, we used a general result on simulations (Theorem 32), and derived the desired implication from linearizability for observational refinement (Corollary 40). A similar approach was taken when we showed that sequential consistency implies observational refinement. (See Definition 33, Theorem 42, and Corollary 43.)

Interestingly, to establish soundness in the general case, we do not need to change the definition of either the weak equivalence relation or linearizability relations. It suffices to change the way these relations are lifted from histories to object systems.[12] Specifically, instead of lifting these relations only to quiescent object systems, as was done in Definition 31, we can lift them to (possibly) non-quiescent object systems by, essentially, ignoring the non-quiescent histories in the object systems.

[12] Note that the weak equivalence relation, the linearizability relation, and, in general, any simulation relation (Definition 30) are defined as relations between arbitrary (well-formed) histories, which are not necessarily quiescent.

Definition 46 (*Lifted Relations by Restriction*)**.** Let $\mathcal{R}$ be a binary relation on well-formed histories. The **lifting by restriction** of $\mathcal{R}$ to object systems OS_A and OS_C, denoted by $OS_C \overset{\downarrow}{\lhd}_{\mathcal{R}} OS_A$, is

$$OS_C \overset{\downarrow}{\lhd}_{\mathcal{R}} OS_A \iff \mathsf{getQuiescent}(OS_C) \lhd_{\mathcal{R}} \mathsf{getQuiescent}(OS_A).$$

Changing the way we lift relations has no effect on our general approach for proving observational refinement presented in Section 5.2.1: Replacing $\lhd$ with $\overset{\downarrow}{\lhd}$ in Theorem 32 does not require any changes to its proof. A similar replacement extends the definitions of sequential consistency (Definition 36) and linearizability (Definition 37) to allow for non-quiescent object systems. The main results regarding sequential consistency and linearizability, such as Theorems 39 and 42 and their corollaries, remain true even if we drop the restrictions to quiescent object systems in those results and allow all object systems. The proofs of the results in Section 5.2.1 can be reused without any modifications.

Intuitively, the reason that this simple adaptation works is that our notion of observational refinement considers only terminating, completed computations, i.e., ones that can be produced by the trace semantics $T(P)$ of programs P in Fig. 5. (See the proof of Theorem 32.) As we have already noted, the projection of such traces on object actions results in quiescent histories.

5.3.2. Completeness

Replacing $\lhd$ with $\overset{\downarrow}{\lhd}$ in Theorems 44 and 45 allows us to establish the desired completeness result without changing the proofs given in Section 5.2.2. Intuitively, this simple adaptation works because (i) our notion of observational refinement considers only terminating, completed computations, and (ii) the lifting operator $\overset{\downarrow}{\lhd}$ ignores non-quiescent histories.

While formally correct, the second reason in the argument above does not seem faithful to the original intentions of Herlihy and Wing [12]. Definition 46 does not place any restrictions on the non-quiescent histories in the object systems, whereas Herlihy and Wing imposed a condition on non-quiescent histories: Every non-quiescent history H in OS_C should correspond to a quiescent history in OS_A that can be obtained by completing some pending calls in H with arbitrary responses and dropping the remaining pending calls in H. Stated differently, every non-quiescent history H in OS_C should correspond to a terminating history in OS_A. Below, we define another lifting of relations, which is more in line with the spirit of [12].

Definition 47 (*Completable Histories*)**.** Given a well-formed history H, we denote by $complete(H)$ the maximal subsequence of H which contains no pending invocations. We say that a quiescent well-formed history H' is a **completion** of H if $H' = complete(H; H_{res})$, where H_{res} is a (possibly empty) sequence of response actions. We denote the set of all possible completions of H by $completions(H)$.

We now try to use the notion of completable histories to modify the definition of lifted relations.

Definition 48 (*Lifted Relations by Completion*)**.** Let $\mathcal{R}$ be a binary relation on well-formed histories. The **lifting by completion** of $\mathcal{R}$ to object systems OS_A and OS_C, denoted by $OS_C \overset{c}{\lhd}_{\mathcal{R}} OS_A$, is

$$\begin{aligned} OS_C \overset{c}{\lhd}_{\mathcal{R}} OS_A \quad \iff \quad & \forall H \in OS_C.\ \exists H_c \in completions(H). \\ & \exists H' \in \mathsf{getQuiescent}(OS_A).\ (H_c, H') \in \mathcal{R}. \end{aligned}$$

Note that if H is a quiescent history then $H = complete(H)$. Also note that $\lhd$, $\overset{\downarrow}{\lhd}$, and $\overset{c}{\lhd}$ agree on the way the lifted $\mathcal{R}$ should behave concerning quiescent histories.

Unfortunately, replacing $\lhd$ with $\overset{c}{\lhd}$ in Theorems 44 and 45 does not allow us to establish the desired completeness result without changing the proofs given in Section 5.2.2. Intuitively, the reason is that (i) our notion of observational refinement considers only terminating, completed computations, and (ii) considering $H_c \in completions(H)$ instead of H in the lifted relation does not add new behaviours to OS_C. Finding a proper remedy for this issue is a future work.

6. Abstraction-based characterization

Section 5 gives a complete characterization of sequential consistency and linearizability. However, it does not explain where the relations $\equiv$ and $\sqsubseteq$ in sequential consistency and linearizability come from. In this section, we answer this question using an abstraction-based approach.

Our answer complements a standard informal view of sequential consistency and linearizability as conditions ensuring the illusion of atomicity. The answer provides an alternative view where sequential consistency and linearizability are understood on the basis of conservative over-approximations of dependencies among object system actions that may arise in some client programs.

The result of this section is based on a reading of a well-formed history H, where H means not the single trace H itself but the set of all the well-formed traces whose object actions are described by H. Formally, we let *WHist* be the set of all the well-formed histories, and use the function means, which is defined in Section 5.1.

Using means, we define a new relation on well-formed histories, which compare possible dependencies between actions in the histories.

Definition 49 (*Abstract Dependency*). For each well-formed history H, the **abstract dependency** $<^{\#}_{H}$ for H is the binary relation on actions in H determined as follows:

$$H_i <^{\#}_{H} H_j \iff i < j \wedge \exists \tau \in \mathsf{means}(H).\ H_i <^{+}_{\tau} H_j.$$

Definition 50 (*Causal Complexity Relation*). The **causal complexity relation** $\sqsubseteq^{\#}$ is a binary relation on well-formed histories, such that $H \sqsubseteq^{\#} S$ iff there exists a bijection $\pi : \{1, \ldots, |H|\} \to \{1, \ldots, |S|\}$ satisfying (i) $\forall i \in \{1, \ldots, |H|\}.\, H_i {=} S_{\pi(i)}$ and (ii) $\forall i, j \in \{1, \ldots, |H|\}.\, H_i <^{\#}_{H} H_j \Longrightarrow S_{\pi(i)} <^{\#}_{S} S_{\pi(j)}$.

Intuitively, $H \sqsubseteq^{\#} S$ means that S is a rearrangement of actions in H that preserves all the abstract causal dependencies in H. Note that S might contain abstract causal dependencies that are not present in H.

The results below show when sequential consistency or linearizability coincides with causal complexity relation.

Theorem 51. *If client operations of different threads are independent, we have that*

$$\forall H, S \in \mathit{WHist}.\ \ H \equiv S \iff H \sqsubseteq^{\#} S.$$

Proof. For each well-formed history H, define a relation $<'_{H}$ on actions of H by

$$H_i <'_{H} H_j \iff (i < j \wedge \mathsf{getTid}(H_i) = \mathsf{getTid}(H_j)).$$

We will prove this theorem by showing two lemmas on well-formed histories. The first lemma is that $H \equiv S$ iff there exists a bijection π such that

$$\forall i, j \in \{1, \ldots, |H|\}.\quad H_i {=} S_{\pi(i)} \wedge (H_i <'_{H} H_j \Longrightarrow S_{\pi(i)} <'_{S} S_{\pi(j)}). \tag{9}$$

The second lemma is that for all well-formed histories H, the two relations $<'_{H}$ and $<^{\#}_{H}$ coincide. Note that the second lemma allows us to replace $<'_{H}$ and $<'_{S}$ in (9) by $<^{\#}_{H}$ and $<^{\#}_{S}$. This replacement would change the first lemma to the equivalence claimed in this theorem. Thus, it is sufficient to prove these two lemmas.

To show the only-if direction of the first lemma, suppose that $H \equiv S$. Then, $|H| = |S|$ and $H|_t = S|_t$ for all thread-ids t. Thus, we can define a bijection $\pi : \{1, \ldots, |H|\} \to \{1, \ldots, |S|\}$ by setting $\pi(i)$ to j where H_i and S_j are the same k-th action in $H|_t$ ($=S|_t$) for some k and t. It is straightforward to show that π is the required bijection in the first lemma. For the if direction, suppose that π is a bijection satisfying (9). Choose an arbitrary thread-id t. We need to show that $H|_t = S|_t$. Since S is a rearrangement of actions in H, we have $|H|_t| = |S|_t|$.[13] Pick an index k of $H|_t$. It suffices to show that $(H|_t)_k = (S|_t)_k$. Let i be the index of H such that H_i is the k-th element of $H|_t$. Then, in the history H, exactly $(k-1)$-many actions by the thread t appear before i, and $(|H|_t| - k)$-many actions by t appear after i. Now, by the implication in (9), in the history S, at least $k-1$ actions by the thread t should appear before $\pi(i)$, and at least $|H|_t| - k$ actions by t should appear after $\pi(i)$. But, $|S|_t| = |H|_t|$. Thus, $\pi(i)$ is the k-th action by the thread t in S. Note that by the equality in (9), $H_i = S_{\pi(i)}$, so that the k-th action of $H|_t$ is the same as the k-th action of $S|_t$.

Now, we move on to the second lemma: $<^{\#}_{H} = <'_{H}$. Note that the inclusion $<^{\#}_{H} \subseteq <'_{H}$ is already proved in Lemma 41. To prove the other inclusion, suppose that $H_i <'_{H} H_j$. Then, $i < j$ and $\neg(H_i \# H_j)$. Thus, $H_i <^{+}_{H} H_j$. Furthermore, since H is well-formed, it belongs to $\mathsf{means}(H)$. By combining $H_i <^{+}_{H} H_j$ and $H \in \mathsf{means}(H)$, we can obtain $H_i <^{\#}_{H} H_j$ as desired. $\square$

Theorem 52. *Assume that for every pair (t, t') of thread-ids with $t \neq t'$, there exist client operations $a \in Cop_t$ and $a' \in Cop_{t'}$ with $\neg(a\#a')$. Under this assumption, we have the following equivalence: $\forall H, S \in \mathit{WHist}.\ H \sqsubseteq S \iff H \sqsubseteq^{\#} S$.*

Proof. For each well-formed history H, define a relation $<''_{H}$ on actions of H by

$$H_i <''_{H} H_j \iff (i < j \wedge (\mathsf{getTid}(H_i) {=} \mathsf{getTid}(H_j) \vee H_i \prec_H H_j)). \tag{10}$$

As in the proof of Theorem 51, we will prove this theorem by showing two lemmas on well-formed histories. The first lemma is that $H \sqsubseteq S$ iff there is a bijection π on $\{1, \ldots, |H|\}$ such that

$$\forall i, j \in \{1, \ldots, |H|\}.\quad H_i {=} S_{\pi(i)} \wedge (H_i <''_{H} H_j \Longrightarrow S_{\pi(i)} <''_{S} S_{\pi(j)}). \tag{11}$$

The second lemma is that for all well-formed histories H, the two relations $<''_{H}$ and $<^{\#}_{H}$ coincide. To see how the conclusion of the theorem follows these lemmas, note that the second lemma allows us to replace $<''_{H}$ and $<''_{S}$ in (11) by $<^{\#}_{H}$ and $<^{\#}_{S}$. This replacement would give the desired equivalence for this theorem. In the remainder of the proof, we will show these two lemmas.

To show the only-if direction of the first lemma, suppose that $H \sqsubseteq S$. Then, there is a bijection π such that

$$\forall i, j \in \{1, \ldots, |H|\}.\quad H_i {=} S_{\pi(i)} \wedge (H_i \prec_H H_j \Longrightarrow S_{\pi(i)} \prec_S S_{\pi(j)}). \tag{12}$$

We will show that π satisfies (11). Suppose that $H_i <''_{H} H_j$ for some i, j. Then, $i < j$. Let $t = \mathsf{getTid}(H_i)$ and $t' = \mathsf{getTid}(H_j)$. We do the case analysis on $H_i \prec_H H_j$. If $H_i \prec_H H_j$, (12) implies that $S_{\pi(i)} \prec_S S_{\pi(j)}$, which in turn entails that $\pi(i) < \pi(j)$ (by

[13] $|H|_t|$ and $|S|_t|$ denote the length of the sequences $H|_t$ and $S|_t$, respectively.

the definition of $\prec_S$). Thus, in this case, we have $S_{\pi(i)} <''_S S_{\pi(j)}$ as desired. If $\neg(H_i \prec_H H_j)$, we should have that $t = t'$, because $H_i <''_H H_j$. Thus,

$$\mathsf{getTid}(S_{\pi(i)}) = \mathsf{getTid}(H_i) = t = t' = \mathsf{getTid}(H_j) = \mathsf{getTid}(S_{\pi(j)}).$$

This means that we can complete the only-if direction by showing that $\pi(i) < \pi(j)$. Note that H_i and H_j should be the call and return actions of the same method call, respectively; otherwise, due to the well-formedness of H, we can find a return for H_i and a call for H_j between H_i and H_j, which entails that $H_i \prec_H H_j$, contradicting our assumption $\neg(H_i \prec_H H_j)$. Another fact to note is that since H is well-formed and H_j is the return for H_i,

$$\forall k.\ \mathsf{getTid}(H_k) = t \Longrightarrow (k < i \Longrightarrow H_k \prec_H H_i) \wedge (j < k \Longrightarrow H_i \prec_H H_k).$$

By (12) and the definition of $\prec_S$,

$$\begin{aligned}&\forall k.\ \mathsf{getTid}(H_k) = t\\ &\quad\Longrightarrow (k < i \Longrightarrow \pi(k) < \pi(i)) \wedge (j < k \Longrightarrow \pi(i) < \pi(k)).\end{aligned} \tag{13}$$

Let m be the number of t's actions in H that appears before i and let n be the number of t's actions in H that occurs after j. Then, by what we have just shown, the number of t's actions in H is $n + m + 2$. For the sake of contradiction, suppose that $\pi(j) < \pi(i)$. (They cannot be the same because π is bijective.) Then, because of (13) and this supposition, at least $(m + 1)$-many actions by t occur before $S_{\pi(i)}$ in S. But, among these $m+1$ actions, there are $m/2+1$ return actions, because $H_j = S_{\pi(j)}$ is a return action and the half of the remaining m actions are return actions. Now, due to the well-formedness of S, all these return actions should have matching call actions in S before them, so we can infer that there are $(m + 2)$ actions by t in S before $\pi(i)$. Since n-many actions of t appear in H after j, (13) implies that there are at least n-many t's actions after $\pi(i)$ in S. By collecting all these numbers, we can infer that S has at least $(m + 2) + 1 + n$ actions by t (where 1 comes from $\pi(i)$). Note that this number is greater than $m + n + 2$, the number of t's actions in H. This is contradictory, because $H|_t = S|_t$.

For the if direction of the first lemma, suppose that π is a bijection satisfying (11). To show that $H \equiv S$, we reuse the proof of Theorem 51. The key observation here is that by the definitions of $<'_H$ and $<''_H$, (11) implies (9). Furthermore, while proving Theorem 51, we already showed that (9) implies $H \equiv S$. Thus, $H \equiv S$ holds here as well. We now show that π satisfies the requirement in the definition of linearizability. Suppose that $H_i \prec_H H_j$. Then, $H_i <''_H H_j$ by (10), and $S_{\pi(i)} <''_S S_{\pi(j)}$ by (11). Let $t = \mathsf{getTid}(S_{\pi(i)})$ and $t' = \mathsf{getTid}(S_{\pi(j)})$. We do the case analysis on $t = t'$. Suppose that $t \neq t'$. Then, by (10), $S_{\pi(i)} \prec_S S_{\pi(j)}$, as desired. Now, suppose $t = t'$. In this case, we only need to show that $S_{\pi(j)}$ is not a return for $S_{\pi(i)}$, because that is the only case where $\neg(S_{\pi(i)} \prec_S S_{\pi(j)})$. Since $t = t'$,

$$\mathsf{getTid}(H_i) = \mathsf{getTid}(S_{\pi(i)}) = t = t' = \mathsf{getTid}(S_{\pi(j)}) = \mathsf{getTid}(H_j).$$

Furthermore, $H_i \prec_H H_j$ by the choice of i, j. Thus, H_j is not a return for H_i. This means that H_i is a return and H_j is a call, or there is some action by the thread t between H_i and H_j. In both cases, (11) implies that $S_{\pi(j)}$ is not a return for $S_{\pi(i)}$, as desired.

Now, we move on to the second lemma: $<^{\#}_H = <''_H$. The inclusion $<^{\#}_H \subseteq <''_H$ is already proved in Lemma 38. To prove the other inclusion, suppose that $H_i <''_H H_j$. Then, $i < j$. Let $t = \mathsf{getTid}(H_i)$ and $t' = \mathsf{getTid}(H_j)$. If $t = t'$, we have $\neg(H_i \# H_j)$. This implies that $H_i <^{\#}_H H_j$, because H is well-formed, so it is in $\mathsf{means}(H)$. Now, consider the other case where $t \neq t'$. Then, $H_i \prec_H H_j$. This means that for some indices k, l of H,

$$(i \leq k < l \leq j) \wedge \mathsf{getTid}(H_k) = t \wedge \mathsf{getTid}(H_l) = t' \wedge \mathsf{retAct}(H_k) \wedge \mathsf{callAct}(H_l).$$

We use the assumption of this theorem, and get client operations $a \in Cop_t$ and $a' \in Cop_{t'}$ with $\neg(a\#a')$. Using these a, a', we define τ to be

$$H_1;\ H_2;\ \dots\ H_k;\ (t, a);\ H_{k+1};\ \dots\ H_{l-1};\ (t', a');\ H_l;\ \dots\ H_{|H|}.$$

Since H is well-formed, H_k is a return by t and H_l is a call by t', the trace τ is well-formed as well. Furthermore,

$$H_i \quad <_\tau \quad H_k \quad <_\tau \quad (t, a) \quad <_\tau \quad (t', a') \quad <_\tau \quad H_l \quad <_\tau \quad H_j.$$

(Note that action (t, a) (resp. (t', a')) is never plugged into the trace in between a matching call and return action of thread t (resp. t'). Thus, it can be generated by having the assumed action a (resp. a') follow (resp. precede) the method invocation using sequential composition.) This shows that $H_i <^+_\tau H_j$. From this follows the desired conclusion: $H_i <^{\#}_H H_j$. □

7. Discussion and related work

Our soundness results exploit the fact that our semantics of programs cannot distinguish one trace from its dependency-preserving permutations. This fact was noticed in the early days of concurrency research, and it was formalized in the various partial order semantics of concurrency [29,23,18]. Indeed, the notions of independent actions and trace equivalence in Section 5.1 are from the theory of Mazurkiewicz traces [18].

However, there is one major difference between our semantics and other partial order semantics: in our semantics two events in a single trace can be related in three ways—definitely dependent, definitely concurrent or unrelated. On the other hand, in classic partial order semantics, the second and third options are usually combined into one, meaning "possibly concurrent". Recall that our semantics splits the call and the return of each method invocation. Thus, when each method call and corresponding return is considered as a single event, traces in our semantics can specify both dependency and concurrency relationships among events explicitly, the first by the dependency relation $<_\tau^+$ and the second by the overlapping call–returns of method invocations. Furthermore, these dependency and concurrency relationships are preserved when traces are combined in the semantics, say, by the interleaving operator, and this preservation result plays a crucial role in our soundness results. This contrasts with the standard approach in partial order semantics, such as Mazurkiewicz traces, where only the dependency relationship is specified explicitly and the concurrency among events is represented by the absence of dependency. This implicit concurrency relationship is not preserved by these semantics. For instance, the parallel composition operator for Mazurkiewicz traces can introduce new dependencies between existing events, thereby invalidating implicit concurrency relationships among those events.

Our soundness proof relies on "definite concurrency" crucially. Roughly speaking, "definite concurrency" prevents clients from creating arbitrary causal relationship between two method invocations. Hence, it allows us to have sound methods for proving observational refinement.

We remark that the splitting of a single event into beginning and ending sub-events, utilized in Herlihy and Wing's definition of linearizability [12], also appeared in the grainless semantics of Reynolds [25] and Brookes [4]. There they used the splitting for a different goal, which is to show that the meanings of *race-free* programs are independent of the atomicity assumption made by programming languages. Note that the correctness conditions for *racy* data structures are the subject of this paper.

Our definition of histories is similar to that of Herlihy and Wing [12], up to a few syntactical differences. However, our notion of object systems is more general than their notion of concurrent objects. There, every object is specified separately and thus, in our terminology, object systems are local. Furthermore, the specification of every object is expected to be done using only sequential histories. While restricting objects to having sequential behaviour allows using existing specification methods to describe the behaviour, we found that, formally, this is not needed. Thus, we do not require that.

De Francesco et al. studied the connection [8,26] between Petri nets and *serializability*, a commonly used and important correctness condition for concurrent database systems [20,28,3]. Informally, a concurrent database system is said to be serializable if for every permitted interleaved execution of transactions (which consist of several atomic steps), one can find equivalent sequential execution, by swapping independent atomic operations of those transactions. Notice that serializability is concerned with entire systems, not libraries or objects or other components that are intended to be used inside bigger enclosing programs. On the other hand, the main concern of linearizability is those libraries and objects. As our results show, linearizability implies a refinement relationship that holds whenever an object is placed in *any* enclosing system. The results of De Francesco et al., concerned as they are with serializability, are not directly applicable for showing the connection between linearizability and observational refinement, as we did in this paper. However, they might provide new insights for proving that the implementations of concurrent libraries are linearizable, since concurrent libraries are often designed from sequential ones by relaxing synchronization requirements.

In this paper, we have not considered the issue of obtaining object systems (sets of histories) from concrete implementations of concurrent objects. One good place to start to address this issue would be the work of Jeffrey and Rathke [15], where they provided a fully abstract semantics of concurrent object systems and where they also considered concurrent objects with features, such as callbacks, more flexible than those assumed in the work on linearizability and this paper.

Shared memory consistency models have been extensively studied by the concurrent-algorithm community. (See, e.g., [27]). Of particular interest to this paper are the works of Graf [9] and Jonsson et al. [16]. These works appeared in a special issue dedicated to various approaches to verifying sequential consistency of the lazy caching algorithm of Afek et al. [1]. While the problem addressed in these works is different and concerns, in our terminology, verifying that the data structure (i.e., the algorithm) produces only sequentially consistent histories, it is still interesting to consider our abstraction-based approach, discussed in Section 6, in the light of the abstract interpretation-based [6] approach of Graf [9]. It is also interesting to compare our simulation-based approach, discussed in Section 5.1, with the transducers-based approach of Jonsson et al. [16]. The latter provides an interesting discussion regarding limitations of the simulation approach in certain cases, e.g., when the implementation has a certain non-deterministic flavour. We can utilize simulation because, unlike [16], we discuss the ramifications of sequential consistency and linearizability from the point of view of the client. In particular, we have a complete record of the history of the interaction between the client program and the data structure.

8. Conclusions

Developing a theory of data abstraction in the presence of concurrency has been a long-standing open question in the programming language community. In this paper, we have shown that this open question can be attacked from a new perspective, by carefully studying correctness conditions proposed by the concurrent-algorithm community, using the tools of programming language theory. We prove that linearizability is a sound method for proving observational refinements for concurrent objects, which is complete when threads are allowed to access shared global variables. When client operations of different threads are independent, we have shown that sequential consistency becomes a sound and complete proof method for observational refinements. We hope that our new understanding on concurrent objects can facilitate the long-delayed transfer of the rich existing theories of data abstraction [13,14,24,19,22] from sequential programs to concurrent ones.

In the paper, we used a standard assumption on the programming language from the concurrent-algorithm community. We assumed that the programming language did not allow callbacks from concurrent objects to client programs, that all the concurrent objects were properly encapsulated [2], and that programs were running under "sequentially consistent" memory models. Although widely used by the concurrent-algorithm experts, these assumptions limit the applicability of our results. In fact, they also limit the use of linearizability in the design of concurrent data structures. Removing these assumptions and extending our results is what we plan to do next.

Acknowledgements

We thank the anonymous referees of this paper and the anonymous referees of the ESOP'09 paper [7] for helping us to improve this work; Viktor Vafeiadis and Matthew Parkinson for useful comments; and Ugo Montanari, Julian Rathke and Matthew Hennessy for pointing out related work. We acknowledge the support of the EPSRC. O'Hearn acknowledges the support of a Royal Society Wolfson Research Merit Award.

References

[1] Y. Afek, G. Brown, M. Merritt, Lazy caching, ACM Trans. Program. Lang. Syst. 15 (1) (1993) 182–205.
[2] A. Banerjee, D.A. Naumann, Representation independence, confinement and access control, in: ACM Symposium on Principles of Programming Languages, 2002, pp. 166–177.
[3] P.A. Bernstein, V. Hadzilacos, N. Goodman, Concurrency Control and Recovery in Database Systems, Addison-Wesley, 1987.
[4] S. Brookes, A grainless semantics for parallel programs with shared mutable data, Electron. Notes Theor. Comput. Sci. 155 (2006) 277–307.
[5] S.D. Brookes, A semantics for concurrent separation logic, in: International Conference on Concurrency Theory, 2004, pp. 16–34.
[6] P. Cousot, R. Cousot, Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints, in: ACM Symposium on Principles of Programming Languages, 1977, pp. 238–252.
[7] I. Filipović, P. O'Hearn, N. Rinetzky, H. Yang, Abstraction for concurrent objects, in: European Symposium on Programming, 2009, pp. 252–266.
[8] N.D. Francesco, U. Montanari, G. Ristori, Modelling concurrent accesses to shared data via Petri nets, in: Programming Concepts, Methods and Calculi, 1994, pp. 403–422.
[9] S. Graf, Characterization of a sequentially consistent memory and verification of a cache memory by abstraction, Distrib. Comput. 12 (2–3) (1999) 75–90.
[10] J. He, C.A.R. Hoare, J.W. Sanders, Data refinement refined, in: European Symposium on Programming, 1986, pp. 187–196.
[11] M. Herlihy, N. Shavit, The Art of Multiprocessor Programming, Morgan Kaufmann, 2008.
[12] M. Herlihy, J.M. Wing, Linearizability: a correctness condition for concurrent objects, ACM Trans. Program. Lang. Syst. 12 (3) (1990) 463–492.
[13] C.A.R. Hoare, Proof of correctness of data representations, Acta Inf. 1 (1972) 271–281.
[14] C.A.R. Hoare, J. He, J.W. Sanders, Prespecification in data refinement, Inform. Process. Lett. 25 (2) (1987) 71–76.
[15] A. Jeffrey, J. Rathke, A fully abstract may testing semantics for concurrent objects, Theoret. Comput. Sci. 338 (1–3) (2005) 17–63.
[16] B. Jonsson, A. Pnueli, C. Rump, Proving refinement using transduction, Distrib. Comput. 12 (2–3) (1999) 129–149.
[17] L. Lamport, How to make a multiprocessor computer that correctly executes multiprocess programs, IEEE Trans. Comput. 28 (9) (1979) 690–691.
[18] A.W. Mazurkiewicz, Traces, histories, graphs: Instances of a process monoid, in: International Symposiums on Mathematical Foundations of Computer Science, 1984, pp. 115–133.
[19] J. Mitchell, G. Plotkin, Abstract types have existential types, ACM Trans. Program. Lang. Syst. 10 (3) (1988) 470–502.
[20] C.H. Papadimitriou, The serializability of concurrent database updates, J. ACM 26 (4) (1979) 631–653.
[21] G. Plotkin, LCF considered as a programming language, Theoret. Comput. Sci. 5 (1977) 223–255.
[22] G. Plotkin, M. Abadi, A logic for parametric polymorphism, in: International Conference on Typed Lambda Calculi and Applications, 1993, pp. 361–375.
[23] V. Pratt, The pomset model of parallel processes: unifying the temporal and the spatial, in: Seminar on Concurrency, 1984, pp. 180–196.
[24] J.C. Reynolds, Types, abstraction and parametric polymorphism, in: R.E.A. Mason (Ed.), Information Processing '83, North-Holland, Amsterdam, 1983, pp. 513–523.
[25] J.C. Reynolds, Toward a grainless semantics for shared-variable concurrency, in: Foundations of Software Technology and Theoretical Computer Science, 2004, pp. 35–48.
[26] G. Ristori, Modelling systems with shared resources via Petri nets, Ph.D. thesis, Dipartimento di Informatica, University of Pisa, 1994.
[27] R.C. Steinke, G.J. Nutt, A unified theory of shared memory consistency, J. ACM 51 (5) (2004) 800–849.
[28] G. Weikum, G. Vossen, Transactional Information Systems: Theory, Algorithms, and the Practice of Concurrency Control, Morgan Kaufmann, 2001.
[29] G. Winskel, M. Nielsen, Models for concurrency, in: Handbook of Logic in Computer Science, Oxford University Press, 1995, pp. 1–148.