On abstraction and compositionality for weak-memory linearisability *

Brijesh Dongol[1], Radha Jagadeesan[2], James Riely[2], and Alasdair Armstrong[1,3]

[1] Brunel University of London, UK
[2] DePaul University, Chicago, USA
[3] University of Cambridge, Cambridge, UK

Abstract. Linearisability is the de facto standard correctness condition for concurrent objects. Classical linearisability assumes that the effect of a method is captured entirely by the allowed sequences of calls and returns. This assumption is inadequate in the presence of relaxed memory models, where *happens-before* relations are also of importance.
In this paper, we develop hb-linearisability for relaxed memory models by extending the classical notion with happens-before information. We consider two variants: *Real time* hb-*linearisability*, which adopts the classical view that time runs on a single global clock, and *causal* hb-*linearisability*, which eschews real-time and is appropriate for systems without a global clock. For both variants, we prove *abstraction* (so that programmers can reason about a client program using the sequential specification of an object rather than its more complex concurrent implementation) and *composition* (so that reasoning about independent objects can be conducted in isolation).

1 Introduction

An implementation is linearisable [19] if for every history of the implementation, there exists a legal history of the specification such that (1) each thread makes the same method invocations in the same order, and (2) the order of non-overlapping invocations is preserved. This notion of linearisability intuitively ensures that each method invocation takes effect between its invocation and response events. Thus, instead of complex concurrent reasoning that requires a characterisation of all possible interactions across method invocations, linearisability ensures that every method call can be understood in isolation via preconditions and postconditions, as familiar in sequential computing.

Linearisability is a local property. Thus, we can reason *compositionally* about a system; i.e., to prove the linearisability of the whole, it suffices to prove the linearisability of projections to components with disjoint memories. This ability to decompose large linearisability proofs is critical to scale the use of linearisability in concurrent reasoning.

* This work was partially supported by EPSRC grants EP/N016661/1 and EP/K008528/1, and NSF Grant No. 1617175.

In this paper, we are also interested in understanding *abstraction* or *contextual refinement* for correctness conditions, i.e., what correctness conditions allow a concurrent object implementation CS to be substituted for an abstract specification AS within a client program? Specifically, we would like to find a condition Z between AS and CS for which

$$Z(AS, CS) \Rightarrow \forall C : Client.\ C[AS] \sqsubseteq C[CS] \qquad (\dagger)$$

holds, where $\sqsubseteq$ denotes some notion of *refinement*, $C[AS]$ denotes a client C that uses the abstract object AS, and $C[CS]$ denotes a client C that uses the implementation object CS.

There are different solutions to (†), depending on the notion of refinement that one uses. Filipović et al. [16] study contextual refinement for terminating computations. They establish refinement between the initial and final states of $C[AS]$ and $C[CS]$ by showing that (†) holds if Z is instantiated to *linearisability*. Others have studied contextual refinement for *traces* of $C[AS]$ and $C[CS]$, where Z must be strengthened to cope with liveness properties [15, 17, 29, 22].

The works cited above assume that threads communicate via *sequentially consistent* (sc) memory [21], where memory events appear to occur according to a single, global, total order consistent with program order. However, high-performance multicore systems typically implement *relaxed* memory models, where memory events may appear to occur out-of-order with respect to program order [1, 2, 5, 6, 23, 25–27]. Under sc, client memory events that occur before a method call in program order cannot overlap with client memory events that occur after the method call (in program order). Under relaxed memory, this property fails to hold.

The impact of relaxed memory in the specification of concurrent data structures is already seen in practice via the explicit specification of *happens-before* (hb) information, e.g., consider the ConcurrentQueue in java.util.concurrent[4]. In addition to the usual guarantee — "The ConcurrentLinkedQueue class supplies an efficient scalable thread-safe non-blocking FIFO queue" — the specification of this class also describes *memory consistency effects* — "As with other concurrent collections, actions in a thread prior to placing an object into a ConcurrentLinkedQueue *happen-before* actions subsequent to the access or removal of that element from the ConcurrentLinkedQueue in another thread." This pattern is repeated for all the classes in this package.

In this paper, we study correctness criteria for data structures in the presence of such memory effects.

– We provide a formalisation of sequential specifications that have been augmented with happens-before information. This augmentation is essential for compositional reasoning. Classical linearisability does not mention *happens-before* information (since it assumes sc). Following [20], we demonstrate by

[4] This package contains data structures and utilities for concurrent programming in Java. See `https://docs.oracle.com/javase/9/docs/api/index.html?java/util/concurrent/package-summary.html`.

example that the traditional perspective fails to ensure contextual refinement for threads that communicate via relaxed memory.
- We define *real-time* hb-*linearisability*, strengthening linearisability to preserve happens-before.
- Concurrent programming in Java-like languages eschews a notion of *global* time. Rather, the idea is that a programmer specifies ordering constraints by defining how threads communicate with explicit mechanisms, such as locks, to ensure that actions taken by one thread would be seen in a reasonable way by other threads [23]. We define *causal* hb-*linearisability* to more directly model this partial order perspective on executions.

Real-time hb-linearisability is stronger than classical linearisability. As we shall demonstrate, causal hb-linearisability and real-time hb-linearisability are incomparable.

Both notions of hb-linearisability are defined relative to a memory model. For both, we show that contextual refinement holds for any relaxed memory model that satisfies the axioms[5] of Alglave, Maranget and Tautschnig (AMT) [3], which are summarised in Section 2. One of our key contributions is the enhancement of AMT to account for events arising from method invocations and responses (Sections 4). We show that, under mild assumptions, any linearisable implementation of concurrent collection must already satisfy the extra happens-before required by a specification; thus discharging the additional proof obligations incurred when proving correctness relative to real-time hb-linearisability.

We provide motivational examples in Section 3. The main definitions and results follow in Sections 4 and 5, respectively.

2 Background: AMT axioms

AMT provide an exhaustive study of relaxed memory models in [3]. Impressively, they manage to capture the details of several specific architectures (including TSO, ARMv7 and Power) in a general framework. They provide a list of axioms that are satisfied by all of the architectures they consider. Fortunately, these axioms are sufficient to establish our results. In this section, we describe the core components of this framework; we refer the interested reader to the original paper [3] for further details.

Let $\mathbb{E}$ be a set of *events*. Each event e is a tuple consisting of a unique identifier, $\mathsf{id}(e)$, a thread identifier, $\mathsf{thread}(e)$, an *action*, and other data. Actions include *memory actions*, e.g., reads and writes; other actions are architecture dependent, including fences. The axioms take as input six relations over $\mathbb{E}$, which together define an *execution*.

- po (program order), which defines a total order on the events of each thread. Events of different threads are unrelated.

[5] While we state our results relative to the axioms of Alglave et al., the ideas behind real-time and causal hb-linearisability can be applied to any other axiomatic memory model based on partial orders.

– co (coherence order), which is a total order on the writes of each location.
– rf (reads from), which maps writes to reads. Each read must be associated with exactly one write, but a write may map to more than one read.
– ppo (preserved program order), which is a suborder derived from po by removing order between events that commute according to the architecture.
– fences, which relates events in po that are separated by a fence.
– prop (propagation order), which relates writes that must propagate to memory in a particular order.

Program order only relates events of the same thread. All of the other relations come in "standard" and "external" versions, denoted with a final e. For example $\mathsf{rfe} \triangleq \{(w, r) \mid (w, r) \in \mathsf{rf} \wedge \mathsf{thread}(w) \neq \mathsf{thread}(r)\}$ relates reads that see writes from a different thread.

The first three relations are execution specific. The remaining relations are defined by the architecture. Additionally, the axioms use the following derived relations:

– $\mathsf{fr} \triangleq \{(r, w_1) \mid \exists w_0.(w_0, r) \in \mathsf{rf} \wedge (w_0, w_1) \in \mathsf{co}\}$, pronounced "from-read". Here r is a read, and w_1 is a write which must come after r, since r has seen a write that preceded w_1 on the same location.
– $\mathsf{hb} \triangleq \mathsf{ppo} \cup \mathsf{fences} \cup \mathsf{rfe}$ defines "happens-before", ordering events that are causally related.

Various architectures can be defined by instantiating these relations in different ways. On TSO, ppo removes the order between a write and a subsequent read. Thus TSO is defined by setting $\mathsf{ppo} = \mathsf{po} \backslash \mathsf{WR}$, $\mathsf{fences} = \mathsf{mfence}$, $\mathsf{prop} = \mathsf{ppo} \cup \mathsf{rfe} \cup \mathsf{fr}$, where WR is the set of all write-read pairs and mfence orders all memory events before a fence with respect to those after the fence.

Executions must satisfy several sanity conditions. For example, co must be a partial order relating only writes to the same location, which is a total order per location. In addition rf is a relation matching each read to a write with the same value and location. For emphasis, we refer to executions that fulfil these requirements as *sane*.

A sane execution is *valid* if it satisfies the four *AMT axioms*. By (No-Thin-Air), causality cannot be cyclic. By (SC-Per-Location), each location taken separately is sc, where $\mathsf{po\text{-}loc}$ is po restricted to events on the same location. By (Observation), events hidden in the causal past cannot be observed. By (Propagation), writes must be propagated in an order consistent with coherence.

$$\mathsf{acyclic}(\mathsf{hb}) \qquad \text{(No-Thin-Air)}$$
$$\mathsf{acyclic}(\mathsf{po\text{-}loc} \cup \mathsf{co} \cup \mathsf{rf} \cup \mathsf{fr}) \qquad \text{(SC-Per-Location)}$$
$$\mathsf{irreflexive}(\mathsf{fre}; \mathsf{prop}; \mathsf{hb}^*) \qquad \text{(Observation)}$$
$$\mathsf{acyclic}(\mathsf{co} \cup \mathsf{prop}) \qquad \text{(Propagation)}$$

We write relations using both set and arrow notation; thus $(a, b) \in \mathsf{po}$ is synonymous with $a \xrightarrow{\mathsf{po}} b$. We also pun between events and their labels in examples.

3 Linearisability for Weak Memory

The goal of the paper is to distinguish "good" implementations, those that ensure contextual refinement, from "bad" ones, that do not. In this section, we present some examples that motivate real-time and causal hb-linearisability as well as the contextual refinement and compositionality properties that they must guarantee.

3.1 Real-time hb-linearisability

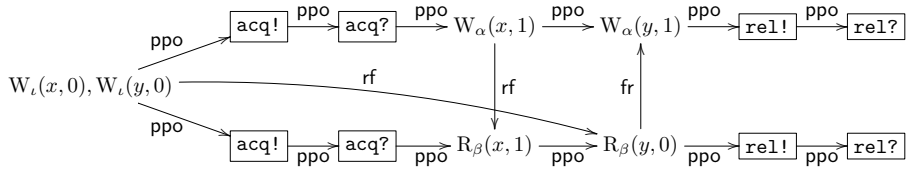
Consider the following code, which uses a lock to coordinate the activities of two threads.

```
Init: x, y = 0, 0
Thread α: lock.acq(); x := 1; y := 1; lock.rel();
Thread β: lock.acq(); print "x=" x; print "y=" y; lock.rel();
```

The abstract specification for a lock will forbid two returns from `acq` without an intervening call to `rel`. From this, a programmer would expect that any execution of the client running against a correctly implemented lock must print `y=1` whenever it prints `x=1`.

For sc-memory, one can establish the validity of this reasoning using classical linearisability. This form of reasoning is unsound, however, for relaxed memory.

To see this, consider the following example, which is a possible execution of the program using the abstract lock specification executing under TSO memory. We extend the AMT model to include events denoting method invocation and response. The invocation of the acquire method is depicted as `acq!`. The return of the acquire method is depicted as `acq?`. The release method is similar.



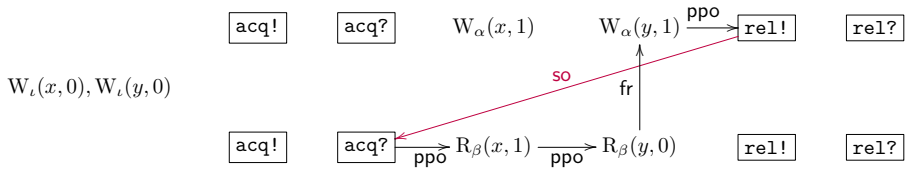
We show the threads in two rows, with events ordered left to right in program order. Applying AMT's framework results in the set of ppo, rf and fr edges between memory events shown above. Since AMT do not address abstract method calls, there are no edges into and out of invocation and response events, except ppo. TSO only relaxes program order between a write and subsequent read. Hence, in the execution above, all program order between memory events is preserved. Here, we have also preserved program order into and out of the method events — in the next subsection and throughout the paper, we consider other strategies for handling the interaction of memory and method events.

The execution satisfies all four AMT axioms, and hence is accepted as a valid execution. However, it clearly violates a programmer's intuition since it describes an execution that prints `x=1` and then `y=0`.

Of course the problem here is that the abstract method calls are being ignored by the memory-model axioms. To remedy this, we must introduce additional

happens-before order that is derived from the abstract specification.[6] We specify this using a *specification order*, so, which relates each invocation to the responses that must *happen-after* that invocation.

If `rel!` in thread α occurs before `acq?` in thread β, the programmer should be able to assume that any event that precedes `rel!` (in program order) must happen before any event that follows the corresponding `acq?`. This should invalidate the execution above. We repeat that execution below, augmented with so, showing only the edges relevant to invalidating this execution.



With hb extended to include so, the exhibited cycle above contradicts AMT's Observation axiom, assuming ppo between a memory and method event is included in prop. Thus, the execution is considered invalid, as desired.

In Section 4, we formalise the concept of a specification augmented with happens-before information and describe its effect on a program execution. We also define real-time hb-linearisability, which is an extension of linearisability that allows one to distinguish a good implementation of an augmented specification from a bad one. A good implementation of the lock must be able *guarantee* the happens-before relation required by the specification. To reason about such implementations, we must also enrich the semantics of implementations to relate method invocation/response events with memory events.

3.2 Causal hb-linearisability

Real-time hb-linearisability is appropriate for tightly coupled systems, such as current generation multicore processors. In distributed systems, however, the cost of real-time synchronisation is high, making real-time hb-linearisability unattractive. We propose *causal* hb-*linearisability* as an alternative notion that requires less synchronisation overhead. Causal hb-linearisability may be appropriate for future generation multicore processors: as the number of cores increases, the necessary synchronisation overhead may force these systems to adopt a looser model (c.f. [28]).

Linearisability requires that the order of non-overlapping methods be preserved in the specification. In the literature, this constraint is motivated by showing that compositionality fails if one requires only that the order of method calls in each thread be preserved. We reexamine these examples in order to motivate causal hb-linearisability.

[6] In fact, APIs such as `java.util.concurrent` document the happens-before behaviour of the methods using edges from the beginning of one method activation to the end of another (or a set of others); that is, from call to return.

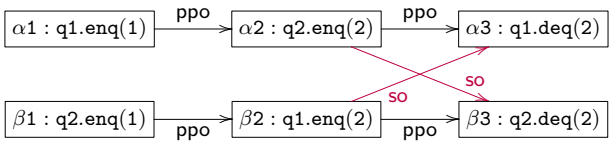


Fig. 1. Non-compositional execution

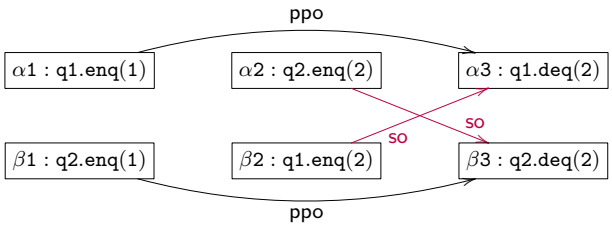


Fig. 2. Compositional execution

Fig. 1 shows a well-known example [18], with method invocation and response collapsed into a single atomic event (shown within a box) for simplicity. Here, threads α and β interact via a pair of queues. The queue specification naturally imposes hb order between an enqueue and a subsequent dequeue of the same element; therefore, the figure shows so edges between $\alpha 2$ and $\beta 3$, as well as between $\beta 2$ and $\alpha 3$. In addition, the figure shows the preserved program order (ppo) between the calls on the two queues in each thread. Recall from Section 2 that $\mathsf{ppo} \subseteq \mathsf{hb}$.

If we consider either $\mathtt{q1}$ and $\mathtt{q2}$ in isolation, the execution is linearisable, since the second enqueue operation for each queue can be considered to have taken effect first. However, the order for each queue is impossible given the order *between* queues. In particular, due to the hb edges when restricting the execution to a single queue and the FIFO ordering requirement of a queue specification, the order of operations for $\mathtt{q1}$ must be $\beta 2 \xrightarrow{\mathsf{hb}} \alpha 1 \xrightarrow{\mathsf{hb}} \alpha 3$, while the order for $\mathtt{q2}$ must be $\alpha 2 \xrightarrow{\mathsf{hb}} \beta 1 \xrightarrow{\mathsf{hb}} \beta 3$. In the full trace, we get a cycle $\alpha 1 \xrightarrow{\mathsf{hb}} \alpha 2 \xrightarrow{\mathsf{hb}} \beta 1 \xrightarrow{\mathsf{hb}} \beta 2 \xrightarrow{\mathsf{hb}} \alpha 1$.

Herlihy and Wing solve this problem by *strengthening* the definition to require that linearisability preserve real-time order of non-overlapping method calls. Thus, the execution of at least one queue in Fig. 1 must be invalidated.

An alternative is to *weaken* ppo to remove the order between events on independent queues. This is analogous to the way events on independent variables are handled under the ARM and Power memory models. Note that we are free to make such a choice here, outside of the implementation memory model, since the ppo order here is at the abstract level between method events. The result is shown in Fig. 2. Here, $\alpha 1$ and $\alpha 3$ are hb ordered, but $\alpha 2$ is not ordered with respect to either event. In this case composition holds. We formalise this intuition as *causal* hb*-linearisability* in Section 4.

We prove abstraction for both real-time and causal hb-linearisability. The story for composition is more complex since clients may be obtrusive, enforcing additional program order between events on different objects. Formally, an execution is *unobtrusive* if it matches a specification string v such that whenever $v = s{\cdot}a\texttt{!}{\cdot}a\texttt{?}{\cdot}t{\cdot}b\texttt{!}{\cdot}b\texttt{?}{\cdot}u$ and $a\texttt{?} \xrightarrow{\mathsf{hb}} b\texttt{!}$ in the execution then $a\texttt{!} \xrightarrow{\mathsf{so}} b\texttt{?}$ in the specification. A client is unobtrusive if all of its executions are unobtrusive. An obtrusive client, such as the one in Fig. 1, may place a fence between the method calls, or use some form of synchronisation to enforce order between them, for example by writing-to and then reading-from another thread. An unobtrusive client, such as the one in Fig. 2, must perform no such synchronisation. We show that causal hb-linearisability satisfies compositionality if the client is unobtrusive.

Obtrusive clients are not problematic if the specification is *commutative*, i.e., if for any specification string $s \cdot a\texttt{!} \cdot a\texttt{?} \cdot t \cdot b\texttt{!} \cdot b\texttt{?} \cdot u$ either $a\texttt{!} \xrightarrow{\mathsf{so}} b\texttt{?}$ or $s \cdot b\texttt{!} \cdot b\texttt{?} \cdot t \cdot a\texttt{!} \cdot a\texttt{?} \cdot u$ is a specification string. Figs. 1 and 2 show the interaction of a client with a composite double-queue. A double-queue specification is not commutative because it does not permit reordering of calls to `enq`, yet the usual specification does not contain a happens-before specification among calls to `enq`. An example of a commutative specification is a double *bag* or *multi-set* where a call to `add` is specified to happen-before the corresponding `remove`, but where all commutations are permitted between the operations on separate elements. We show that causal hb-linearisability satisfies compositionality if the specification is commutative.

4 Traces and Weak-Memory Semantics

In this section we formalise the interaction between a *client* and a set of *objects*.

We divide the event set $\mathbb{E}$ into four disjoint subsets: Let $\mathbb{C}$ be the set of *client events*, let $\mathbb{O}$ be the set of *object events*, let $\mathbb{I}$ be the set of *invocation events* and let $\mathbb{R}$ be the set of *response events*. We use $\mathbb{M} \subseteq \mathbb{I} \cup \mathbb{R}$ to range over *method events*. Like others [15–17], we assume clients and objects only communicate via the object interface, specified as subset of $\mathbb{I} \cup \mathbb{R}$. Thus, clients and objects must operate over disjoint sets of locations: $\mathsf{location}(e) \neq \mathsf{location}(f)$ for any $e \in \mathbb{C}$ and $f \in \mathbb{O}$.

For any relation $R \subseteq \mathbb{E} \times \mathbb{E}$ and set $\mathbb{X} \subseteq \mathbb{E}$, let $R|\mathbb{X}$ denote the restriction of R to $\mathbb{X}$, i.e., $R|\mathbb{X} = R \cap (\mathbb{X} \times \mathbb{X})$.

In order to connect AMT-style executions to specifications, we work with strings of events, which we refer to as *traces*[7]. While our definitions are given directly in terms of traces, we often use program syntax in examples. It is straightforward to define a semantics which gives the denotation of programs as sets of traces, where memory reads and method returns may yield any value. For example, the semantics of "`x:=1;push(5)`" is the set $\{\mathrm{W}_\alpha(x,1) \cdot \mathit{push}!_\alpha(5) \cdot \mathit{push}?_\alpha \mid \alpha \in \mathit{Threads}\}$. The semantics of "`r1:=pop();r2:=x`" is the set $\{\mathit{pop}!_\beta \cdot$

[7] Since events include unique identifiers (and therefore cannot repeat), there is an isomorphism between strings of events and total orders over finite set of events.

$pop?_\beta(u) \cdot \mathrm{R}_\beta(x,v) \mid \beta \in \mathit{Threads}, u \in \mathit{Values}, v \in \mathit{Values}\}$. The semantics of "`x:=1;push(5) || r1:=pop();r2:=x`" is any interleaving of these where $\alpha \neq \beta$. Note that both the pop and the read of `x` may return any value.

In the remainder of this introductory text, we consider method events and memory events independently: method events relate to specifications and memory events relate to executions. In the following subsections, we show how these can be combined, both for abstract and concrete object systems.

First we discuss method events. Rather than modelling specifications using formal languages such as Larch, Z, or LTL, we model specifications semantically as sets of strings of method events. Strings provide a total order on the method events, which is sufficient to capture sequential behaviours. When considering event strings as specifications, we ignore the thread identifier in events. For example, the specification of stack includes strings such as $\mathit{push}!(5) \cdot \mathit{push}? \cdot \mathit{pop}! \cdot \mathit{pop}?(5)$. Thus a trace of method events may be seen directly as an element of a specification, where we ignore thread identifiers. For example, the trace $\mathit{push}!_\alpha(5) \cdot \mathit{push}?_\alpha \cdot \mathit{pop}!_\beta \cdot \mathit{pop}?_\beta(5)$ is a valid trace for a stack, whereas neither $\mathit{push}!_\alpha(5) \cdot \mathit{push}?_\alpha \cdot \mathit{pop}!_\beta \cdot \mathit{pop}?_\beta(1)$ nor $\mathit{pop}!_\beta \cdot \mathit{pop}?_\beta(5) \cdot \mathit{push}!_\alpha(5) \cdot \mathit{push}?_\alpha$ is valid.

We now discuss memory events. From a trace, we can generate AMT executions as follows.

Definition 1. *A tuple* $(t, \mathsf{co}, \mathsf{rf}, \mathsf{ppo}, \mathsf{fences}, \mathsf{prop})$ *is an* execution *of trace* t *if these relations satisfy AMT's sanity conditions (see Section 2), where program order is given by* $\mathsf{po}_t = \{(e,f) \mid (e,f) \in t \wedge \mathsf{thread}(e) = \mathsf{thread}(f)\}$.

Let $\mathsf{execs}(t)$ *be the set of executions of* t*. We use* τ *to range over executions.*

Note that we require executions to be sane, but do not enforce validity at this stage. We discuss validity in the following subsections. We usually drop subscripts from order relations, preferring po to po_t, etc.

A single trace may give rise to many executions. For example, the program "`r=x || x:=5 || x:=5`" gives rise to a set of traces which includes $\mathrm{R}_\alpha(x,5) \cdot \mathrm{W}_\beta(x,5) \cdot \mathrm{W}_\gamma(x,5)$. Executions of this trace may have either $\mathrm{W}_\beta(x,5) \xrightarrow{\mathsf{rf}} \mathrm{R}_\alpha(x,5)$ or $\mathrm{W}_\gamma(x,5) \xrightarrow{\mathsf{rf}} \mathrm{R}_\alpha(x,5)$. This program also gives rise to traces such as $\mathrm{R}_\alpha(x,1) \cdot \mathrm{W}_\beta(x,5) \cdot \mathrm{W}_\gamma(x,5)$, which has no executions, since the read can not be fulfilled by any write.

The trace order of events from the same thread determines program order; however, the trace order between events from different threads is ignored. If $\alpha \neq \beta$, then the traces $\mathrm{R}_\alpha(x,5) \cdot \mathrm{W}_\beta(x,5)$ and $\mathrm{W}_\beta(x,5) \cdot \mathrm{R}_\alpha(x,5)$ generate exactly the same executions, modulo the trace itself, as do $\mathit{push}!_\alpha(5) \cdot \mathit{push}?_\alpha \cdot \mathit{pop}!_\beta \cdot \mathit{pop}?_\beta(5)$ and $\mathit{pop}!_\beta \cdot \mathit{pop}?_\beta(5) \cdot \mathit{push}!_\alpha(5) \cdot \mathit{push}?_\alpha$.

4.1 Clients with object specifications

We now discuss the semantics of client-object systems, where object behaviours are described by a specification. In terms of condition (†) from the introduction, this section formalises the behaviours of $C[AS]$. As discussed in Section 3, it

is important for specifications to provide happens-before guarantees to client programs to enable writes to propagate in the correct order.

Example 2. Consider the program "`x:=5;push(5) || r1:=pop();r2:=x`". If variable `x` is initialised to 0, then the following is a trace of this program:

$$\mathrm{W}_\iota(x,0) \cdot \mathrm{W}_\alpha(x,5) \cdot \mathit{push}!_\alpha(5) \cdot \mathit{push}?_\alpha \cdot \mathit{pop}!_\beta \cdot \mathit{pop}?_\beta(5) \cdot \mathrm{R}_\beta(x,0)$$

There are valid AMT executions of this trace. Here the thread β returns a value 0 for x, missing the value 5 written by thread α, even if we assume that the memory model guarantees $\mathrm{W}_\iota(x,0) \xrightarrow{\mathsf{hb}} \mathrm{W}_\alpha(x,5)$. We wish to disallow such traces via extra happens-before orders introduced via the stack specification. In particular, we enhance the specification with an additional ordering relation that ensures each *push* is hb ordered before the corresponding *pop*. When used in a client program, we assume that such an enhanced specification induces additional order, namely that it ensures $\mathrm{W}_\alpha(x,5) \xrightarrow{\mathsf{hb}} \mathrm{R}_\beta(x,0)$. Now, if the memory model ensures $\mathrm{W}_\iota(x,0) \xrightarrow{\mathsf{hb}} \mathrm{W}_\alpha(x,5)$ the trace becomes invalid, as intended. □

In order to encode happens-before information, we take a specification string to be pair consisting of a string of method events, h, and a *specification-based happens-before order* so, relating events in h. For example, in the stack specification $h = \mathit{push}!(5) \cdot \mathit{push}? \cdot \mathit{pop}! \cdot \mathit{pop}?(5)$, we expect that $\mathit{push}!(5) \xrightarrow{\mathsf{so}} \mathit{pop}?(5)$. Various choices of so are possible. The Java concurrency APIs specify that each push happens-before the corresponding pop. Many concurrent implementations actually give stronger guarantees, which could be included in the specification if one wished. For example, a Trieber stack guarantees that a push happens-before the matching pop *and* every subsequent pop. If this specification were adopted, the client programmer would be able to make stronger assumptions. Our results are parametrised by a chosen specification.

In the following definition, we recall that $\mathbb{M} \subseteq \mathbb{I} \cup \mathbb{R}$ is the set of method events, and require that so only relate invocations to responses. In addition, so must be consistent with h.

Definition 3 (Specification). *A* specification *is a pair* $(\mathbb{M}, H)$*, where* $H \subseteq 2^{\mathbb{M}\times\mathbb{M}} \times 2^{\mathbb{I}\times\mathbb{R}}$ *such that for each* $(h, \mathsf{so}) \in H$*,* h *is a total order and* $\mathsf{so} \subseteq h$*.*

We now define what it means for a client to interact with an abstract specification: When projected to client events, we must have a sane AMT execution. When projected to method events, we must have a specification string.

Definition 4 (Client-specification execution). *Let* $AS = (\mathbb{M}, H)$ *be a specification,* C *be a client and* $t \in (\mathbb{M} \cup \mathbb{C})^*$ *be a trace of* $C[AS]$*. We say that the tuple* $(t, \mathsf{co}, \mathsf{rf}, \mathsf{ppo}, \mathsf{fences}, \mathsf{prop})$ *is a* client-specification execution for so *iff*

- $(t|\mathbb{C}, \mathsf{co}, \mathsf{rf}, \mathsf{ppo}, \mathsf{fences}, \mathsf{prop}) \in \mathsf{execs}(t|\mathbb{C})$*, where* $t|\mathbb{C}$ *is the trace* t *restricted to elements in* $\mathbb{C}$*, and*
- $\mathsf{so} \subseteq \mathbb{I} \times \mathbb{R}$ *is an order such that* $(t|\mathbb{M}, \mathsf{so}) \in H$*.*

A valid client-specification execution must satisfy the AMT axioms, where method events are included in the happens-before relation by lifting so to relate memory events ordered by $\mathsf{po}; \mathsf{so}; \mathsf{po}$.

Definition 5 (Valid client-specification execution). *A* client-specification execution for so *is* valid *iff the AMT axioms hold with* $\mathsf{hb} \triangleq \mathsf{ppo} \cup \mathsf{fences} \cup \mathsf{rfe} \cup \mathsf{hbs}$*, where*

$$\mathsf{hbs} = \{(e, e') \subseteq \mathbb{C} \times \mathbb{C} \mid \exists i \in \mathbb{I}, r \in \mathbb{R}.\ e \xrightarrow{\mathsf{po}}_t i \wedge i \xrightarrow{\mathsf{so}} r \wedge r \xrightarrow{\mathsf{po}}_t e'\}.$$

4.2 Client-implementation traces

We now describe the meaning of a client that executes with an implementation object. In terms of condition (†) from the introduction, this section formalises the behaviours of $C[CS]$. One can interpret a client interaction with a concrete object system as an execution by simply removing method events. For example, if s is the sequence of memory events implementing push, and t is the sequence of memory events implementing pop, then the following is a concrete trace of the program in Example 2:

$$\mathrm{W}_\iota(x,0) \cdot \mathrm{W}_\alpha(x,5) \cdot \mathit{push}!_\alpha(5) \cdot s \cdot \mathit{push}?_\alpha \cdot \mathit{pop}!_\beta \cdot t \cdot \mathit{pop}?_\beta(5) \cdot \mathrm{R}_\beta(x,0).$$

One could say that this trace is valid exactly if $\mathrm{W}_\iota(x,0) \cdot \mathrm{W}_\alpha(x,5) \cdot s \cdot t \cdot \mathrm{R}_\beta(x,0)$ is valid, i.e., the trace with method events removed. While sufficient for some purposes, any connection with the abstract object system is lost. In this section we describe how to integrate method actions into concrete executions so as to support a notion of operational refinement between concrete and abstract systems.

In general, a terminating thread of a client/object interaction has the form $\mathbb{C}^*(\mathbb{I}\mathbb{O}^*\mathbb{R}\mathbb{C}^*)^*$: the client may perform memory actions in $\mathbb{C}$ until it invokes a method, giving control to the object; the object then may perform memory actions $\mathbb{O}$ until it returns, giving control back to the client[8]. In executions of concrete traces, method events are placeholders, which should have no memory effects themselves. Any memory effects should arise from the concrete implementation. Thus we expect that empty methods should have no effects in a concrete system. More generally, our definition should support method inlining.

The problem boils down to which po edges between method events and memory events should be preserved in ppo, and therefore hb. Since method events denote the boundary between client events and object events, there are two sets of edges to consider:

(**1**) po edges between $\mathbb{M}$ and $\mathbb{C}$, and
(**2**) po edges between $\mathbb{M}$ and $\mathbb{O}$.

We cannot preserve *both* (**1**) and (**2**). To see why, consider the trace t below, where $c, c' \in \mathbb{C}$, $i \in \mathbb{I}$, $r \in \mathbb{R}$ and $o, o' \in \mathbb{O}$ are events of the same thread:

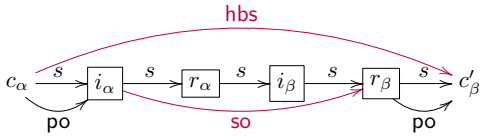
$$c \xrightarrow{t} \boxed{i} \xrightarrow{t} o \xrightarrow{t} \ldots o' \xrightarrow{t} \boxed{r} \xrightarrow{t} c'$$

[8] Recall from the beginning of this section $\mathbb{O}$ and $\mathbb{C}$ range over disjoint sets of memory locations.

The preserved po edges must *not* introduce any new ppo order between c and o that is not present in the memory model since the invocation i, in isolation, cannot affect memory. In other words, if both $c \xrightarrow{\mathsf{po}} i$ and $i \xrightarrow{\mathsf{po}} o$ were preserved, this would ultimately create a transitive happens-before edge between c and o, disallowing them from being reordered even in a memory model that doesn't enforce this restriction. For example, in TSO, we may have $c = W(z, 1)$ and $o = R(x, 2)$, which may be reordered; introduction of a method invocation between c and o should not prevent the reordering from occurring.

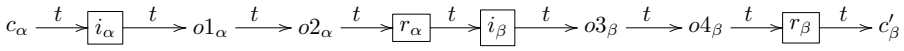
We must also preserve (**1**) and (**2**) in such a way that we are able to decouple object correctness (hb-linearisability) from contextual refinement. Our solution is to *always* preserve (**1**), resulting in a set of edges cio (client-interface order), and *conditionally* preserve (**2**), resulting in a set of edges oio (object-interface order). The intention is to introduce both cio and oio into an extended hb ordering.

To justify our choices, consider the abstract trace s, given below, which shows a client interacting with an abstract specification object. The actions c_α and c_β are client actions of separate threads, α and β.



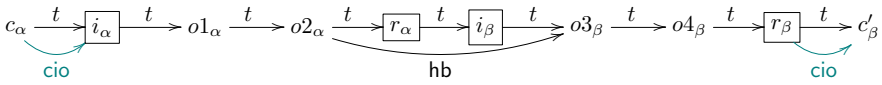
Assume that the specification requires so between i_α and r_β. Thus, by Definition 5, for any execution of s, we must have an hbs edge between client events c_α and c'_β. This (again by Definition 5) means that we have $c_\alpha \xrightarrow{\mathsf{hb}} c'_\beta$ since hbs $\subseteq$ hb.

Suppose we wish to determine whether the trace t below is a *contextual refinement* of s.



Among other things, we must be able to guarantee $c_\alpha \xrightarrow{\mathsf{hb}} c'_\beta$ for any execution of t (see Definition 9) since this order is present in the specification.

An implementation of the sequential object can take us part of the way there by ensuring hb between object events. Suppose for our example that the implementation guarantees $o2_\alpha \xrightarrow{\mathsf{hb}} o3_\beta$. This, together with the fact that we always preserve (**1**), results in an execution of the following form:

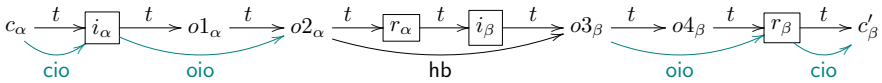


Note that the hb above is introduced via AMT's standard conditions described in Section 2, i.e., without taking invocations and responses into account.

To complete the hb chain from c_α to c'_β, we require edges from i_α to $o2_\alpha$ and from $o3_\beta$ to r_β. The condition for preserving (**2**) is as follows. Suppose o is an object event. We preserve program order from an invocation i to o if after replacing the action in i with an *arbitrarily chosen* memory action to obtain an event i', the relation ppo $\cup$ fences orders i' before o. The case for ordering o

before a response event is similar. Since an arbitrary memory action is ordered before o, any client memory action that occurs before i in t must also be ordered with respect to o. Moreover, these conditions are independent of any specific client, and hence our treatment allows one to reason about the properties of the concurrent object (e.g., hb-linearisability) in isolation, relying on our abstraction theorem to guarantee contextual refinement.

Suppose for our example that $o2_\alpha$ and $o3_\beta$ do indeed satisfy the conditions described above. Our execution thus becomes:



The presence of the oio edge to $o2_\beta$ means that any client memory event of α that precedes i_α in program order must also be ordered with $o2_\beta$ (since an arbitrary action was considered when constructing oio). Since we do not have an oio edge to $o1_\alpha$, it would be possible to reorder c_α with $o1_\alpha$ if the memory model semantics permits the reordering. (The same applies to $o4_\beta$ and c'_β.) Thus, we have only introduced as much order as necessary.

Example 6. Consider TSO, and suppose $c_\alpha = W_\alpha(x,1)$, $o1_\alpha = R_\alpha(y,2)$ and $o2_\alpha = W_\alpha(z,4)$. We do not have $i_\alpha \xrightarrow{\mathsf{oio}} o1_\alpha$ since for TSO, $\mathsf{ppo} = \mathsf{po}\backslash\mathsf{WR}$. However, we do have $i_\alpha \xrightarrow{\mathsf{oio}} o2_\alpha$. The program under TSO could reorder c_α and $o1_\alpha$, but would never reorder c_α and $o2_\alpha$. Now suppose $o1_\alpha = \mathsf{fence}$ and $o2_\alpha = R_\alpha(y,3)$. Again, we have $i_\alpha \xrightarrow{\mathsf{oio}} o2_\alpha$, but in this instance, the order from i_α is generated by the fence.

We now formalise both orders in the context of an *extended execution*, i.e., an execution extended with orders cio and oio. Such a definition is necessary because the definition of oio requires relabelling of invocation actions within a trace. In the definition below, we let $\mathsf{relabel}(e,a)$ denote the event e with its action relabelled to a, $\mathsf{labels}(e) = \{\mathsf{relabel}(e,a) \mid a \text{ is a memory action}\}$ denote the set of all possible relabelings of e and $t[e'/e]$ denote the trace t with event e replaced by e'.

Definition 7 (Client-implementation execution). *For a trace t, we say that the tuple $(t, \mathsf{co}, \mathsf{rf}, \mathsf{ppo}, \mathsf{fences}, \mathsf{prop}, \mathsf{cio}, \mathsf{oio})$ is a* client-implementation execution *iff the $(t|(\mathbb{C}\cup\mathbb{O}), \mathsf{co}, \mathsf{rf}, \mathsf{ppo}, \mathsf{fences}, \mathsf{prop}) \in \mathsf{execs}(t|(\mathbb{C}\cup\mathbb{O}))$, and*

$$\mathsf{cio} = \mathsf{po}_t \cap ((\mathbb{C}\times\mathbb{I}) \cup (\mathbb{R}\times\mathbb{C}))$$
$$\mathsf{oio} = \{(i,o) \in \mathbb{I}\times\mathbb{O} \mid \mathsf{IO}(t,i,o)\} \cup \{(o,r) \in \mathbb{O}\times\mathbb{R} \mid \mathsf{OR}(t,o,r)\}$$

where IO *and* OR *are defined as follows.*

$$\mathsf{IO}(t,i,o) = \forall i' \in \mathsf{labels}(i).\ \forall \tau \in \mathsf{execs}(t[i'/i] \mid (\mathbb{C}\cup\mathbb{O}\cup\{i'\})).\ i' \xrightarrow{\mathsf{ppo}\cup\mathsf{fences}}_\tau o$$
$$\mathsf{OR}(t,o,r) = \forall r' \in \mathsf{labels}(r).\ \forall \tau \in \mathsf{execs}(t[r'/r] \mid (\mathbb{C}\cup\mathbb{O}\cup\{r'\})).\ o \xrightarrow{\mathsf{ppo}\cup\mathsf{fences}}_\tau r'$$

In this definition, $e \xrightarrow{\mathsf{ppo}\cup\mathsf{fences}}_\tau f$ denotes $(e,f) \in \mathsf{ppo}_\tau \cup \mathsf{fences}_\tau$, recalling that ppo_τ and fences_τ are the ppo and fences relations of the execution τ, respectively.

Within $\mathsf{IO}(t, i, o)$, for any i' obtained by replacing the action in i with a memory action, and any execution τ of the trace t with i replaced by i' restricted to $\mathbb{C} \cup \mathbb{O} \cup \{i'\}$, we have that i' is ordered before o with respect to ppo_τ or fences_τ. The predicate $\mathsf{OR}(t, i, o)$ is similar.

Definition 8 (Valid client-implementation execution). *We say client-implementation execution is* valid *iff the AMT axioms hold, where* $\mathsf{hb} \triangleq \mathsf{ppo} \cup \mathsf{fences} \cup \mathsf{rfe} \cup \mathsf{cio} \cup \mathsf{oio}$.

Our notion of contextual refinement is based purely on the observations that a client makes over the memory and object states. Thus, it simply ensures that every valid execution of the client when using the implementation object is a possible execution of the client when it uses the specification object.

Definition 9 (Contextual refinement). *Suppose t is a trace of $C[CS]$ and s is a trace of $C[AS]$. We say t* contextually refines *s (denoted $s \sqsubseteq t$) iff*

- $t|\mathbb{C} = s|\mathbb{C}$,
- *whenever* $(t, \mathsf{co}, \mathsf{rf}, \mathsf{ppo}, \mathsf{fences}, \mathsf{prop}, \mathsf{cio}, \mathsf{oio})$ *is a valid client-implementation execution,* $(s, \mathsf{co}|\mathbb{C}, \mathsf{rf}|\mathbb{C}, \mathsf{ppo}|\mathbb{C}, \mathsf{fences}|\mathbb{C}, \mathsf{prop}|\mathbb{C})$ *is a valid client-specification execution.*

The first condition requires that s and t restricted to client events are equal (i.e., they have the same denotational behaviour), whereas the second requires that a valid execution of t can be restricted to form a valid execution of s. In particular, if t is valid, then s must also be valid.

Contextual refinement is lifted to the level of programs in the standard manner. We say $C[AS]$ is *contextually refined* by $C[CS]$, denoted $C[AS] \sqsubseteq C[CS]$, iff for every valid trace t of $C[CS]$, there exists a valid trace s of $C[AS]$ such that $s \sqsubseteq t$. We say AS is *contextually refined* by CS, denoted $AS \sqsubseteq CS$ iff for any client C, we have $C[AS] \sqsubseteq C[CS]$.

4.3 Implementation objects and happens-before linearisability

In this section, we formalise the correctness expectations on an implementation object in terms of a sequential specification. The notions we develop are based on linearisability. In the context of weak memory, we show that linearisability is not sufficient: additional requirements must be enforced. At the same time, weak memory makes it natural to look at notions of linearisability that do not strictly enforce realtime order. In terms of (†), this section formalises the sorts of behaviours CS must satisfy in order to prove the abstraction property.

Linearisability in a weak memory setting must preserve the happens-before order of an abstract specification. An implementation trace comprises client/object memory events as well as invocation/response events of object operations. From the perspective of an object, invocation/response events abstractly represent a client's memory events in program order. Thus preserved program order between object memory events and invocation/response events are execution specific, and introduced into the happens-before order of a trace.

The final component of hb-linearisability is a restriction on how invocations and responses can be (re)ordered. In a weak memory setting there is more than one potential restriction. Two of these are to: (a) order operation calls according to their real-time program order (real-time hb-linearisability), and (b) order operation calls according to their happens-before order (causal hb-linearisability). Choice (a) is closer to Herlihy and Wing's original definition, while (b) is closer to an ordering one might expect in a weak-memory setting.

In the definition below, like linearisability [19], since operations may take effect before they return, we allow histories to be extended by adding matching responses to operations that have been invoked but not yet returned.

Definition 10. *A valid client execution is* real-time hb-linearisable *with respect to* (h, so) *iff it can be extended to an execution of some trace* t *with happens-before relation* hb *such that the following holds:*

$$\forall \alpha \in \mathit{Threads}.\ [t|\alpha|(\mathbb{I} \cup \mathbb{R}) = h|\alpha] \vee [\exists i \in \mathbb{I}.\ t|\alpha|(\mathbb{I} \cup \mathbb{R}) = (h|\alpha) \cdot i] \qquad \text{(Permutation)}$$

$$\forall i \in \mathbb{I}, r \in \mathbb{R}.\ r \xrightarrow{t} i \Rightarrow r \xrightarrow{h} i \qquad \text{(RTO-Preservation)}$$

$$\forall i \in \mathbb{I}, r \in \mathbb{R}.\ i \xrightarrow{\mathsf{so}} r \Rightarrow i \xrightarrow{\mathsf{hb}} r \qquad \text{(HB-Satisfaction)}$$

An execution is linearisable with respect to a specification $AS = (\mathbb{M}, H)$ *if it is linearisable for some* $(h, \mathsf{so}) \in H$.

Conditions (Permutation) and (RTO-Preservation) are equivalent to Herlihy and Wing's original requirements for linearisability [19]. Thus, hb-linearisability implies standard linearisability. Condition (HB-Satisfaction) ensures that the order between invocations and responses (of different operations) expected by the specification is respected by the happens-before order in the implementation.

Definition 11. Causal hb-linearisability *differs from real-time* hb*-linearisability only in that condition* (RTO-Preservation) *is replaced by:*

$$\forall i \in \mathbb{I}, r \in \mathbb{R}.\ r \xrightarrow{\mathsf{hb}^+} i \Rightarrow r \xrightarrow{h} i \qquad \text{(HB-Preservation)}$$

Real-time and causal hb-linearisability are incomparable since po and hb^+ are incomparable. The differences between these requirements are shown in Figs. 3-6. In Fig. 3, the execution is considered to be sequential according to both conditions, but in the second example, the method calls are causally ordered in a different order to real-time order. In Fig. 4, both executions are concurrent according to real-time order, but sequential according to causal order, in Fig. 5 the operations are considered to be concurrent according to both orders, and in Fig. 6 the execution is real-time sequential, but causally concurrent.

Consider the following simplified version of the queue example from Section 3.2.

Example 12. A two-place buffer has operations $\texttt{put}_1$, $\texttt{put}_2$, $\texttt{get}_1$ and $\texttt{get}_2$. The sequential specification states that a call to $\texttt{get}_i$ must return the argument given on the most recent call to $\texttt{put}_i$. If we follow the model of the data structures in `java.util.concurrent`, the expected happens-before relation is that $\texttt{put}_i$

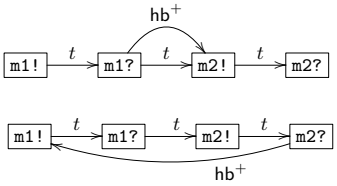


Fig. 3. Real-time and causal sequential

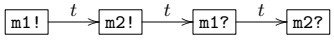


Fig. 5. Real-time and causal concurrent

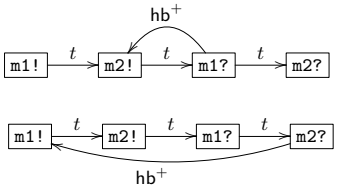


Fig. 4. Real-time concurrent, causal sequential

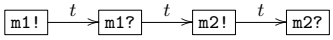


Fig. 6. Real-time sequential, causal concurrent

happens-before any $\texttt{get}_i$ that returns a matching value. In particular, note that there is no happens-before expectation between $\texttt{put}_1$ and $\texttt{get}_2$.

It is possible to implement the two-place buffer using independent synchronisation variables. Supposing that the buffer initially holds zeros, the client

```
Thread α: put1(5); get2()
Thread β: put2(5); get1()
```

can return zero for both calls to `get`. This execution is correct with respect to causal hb-linearisability, but not with respect to real-time hb-linearisability.

We now turn our attention to the cases when the two notions coincide: Real-time hb-linearisability and causal hb-linearisability coincide if for the execution $\tau \in \mathsf{execs}(t)$ under consideration $\mathsf{hb}^+ = t$. In particular, for the sc memory model, real-time and causal hb-linearisability coincide.

We simply use the term hb-linearisability whenever we do not distinguish between real-time and causal hb-linearisability. The next definition lifts hb-linearisability to the level of objects in the standard manner.

Definition 13 (hb-linearisable implementation). *We say that an object CS is an* hb-linearisable implementation *of specification $AS = (\mathbb{M}, H)$ if for all clients C, and all valid executions τ of $C[CS]$, τ is* hb*-linearisable with respect to some $(h, \mathsf{so}) \in H$.*

4.4 Establishing hb-linearisability

In this section, we demonstrate that for some implementations it is no more difficult to establish hb-linearisability than it is to establish standard linearisability. Of course, establishing standard linearisability on a relaxed memory model is still more difficult than under sc memory.

For example, in the Treiber stack algorithm, each method call must perform a compare-and-set operation on a single memory location, representing the top of the stack. The order of successful CAS operations is the linearisation order

used to establish linearisability. It also establishes happens-before between the call of a method and the return of every method that follows it in linearisation order, assuming that CAS is given acquire/release semantics, as in Java and TSO.

We can establish a similar result for any classically linearisable implementation of a collection class, under one of two assumptions:

- there is a memory fence at beginning of every mutator method and at the end of every accessor method, or
- data values are stored in memory locations with acquire/release semantics.

In the remainder of this section, we establish that, in either case, a classically linearisable collection already satisfies the happens-before requirements of the Java collections API.

A *collection class* refers to common data structures such as Stacks, Queues, Lists, Trees that are containers of elements of objects of a given type. In the rest of this discussion, we pick Stack as the example; however, our discussion applies equally well to the other examples. We use the happens-before semantics of the Java collections classes.

Recall that the signature of a `Stack<T>` of elements of type T is given by `void push(T), T pop(), T top()` and `boolean isEmpty()`. The happens-before requirement is carried by the objects of type `T`, i.e. there is a happens-before edge to the return of any `pop` or `top` from the invocation of the corresponding `push`. Notably, there are no happens-before requirement between different push or different pop methods.

Consider an implementation I of `Stack<T>`. We say that I is *generic* if the operations that it performs on values of type `T` are restricted to load, and store. *All* classical Stack (indeed, collection!) algorithms, such as Treiber stack, follow this discipline. We call such implementations generic because such an implementation treats type `T` as abstract, only writing and reading the values, thus eschewing any operation that exploit the structure of type `T`. The correctness of I is implied by studying the traces that restrict the push methods to be of form `push(new T())`, i.e. every push is of a new object reference that has not been seen thus far[9].

In such a restricted trace, it is immediately clear that there is a location that is written by `push(o)` that is also read by a `pop` or `top` method that returns `o`, since the argument to the push is a new reference. Since the AMT axioms ensure that reads-from is always contained in happens-before, this ensures that

[9] The proof of this fact is inspired by proofs of information flow. In an execution trace of I, consider the locations partitioned into "low" locations $\mathcal{L}$ that store values not of type `T` and "high" locations $\mathcal{H}$ that store values of type `T`. Two memories $(\mathcal{L}_1, \mathcal{H}_1)$ and $(\mathcal{L}_2, \mathcal{H}_2)$ are related by $\mathcal{E}$ if they agree on their "low" parts. By the restrictions on generic implementations, any program statement in a generic implementation preserves the $\mathcal{E}$ relationship of memories. Thus, in order to validate I, it suffices to consider the execution traces where pushes are restricted to have new references as parameters.

any classically linearisable generic implementation of Stack will always have the required happens-before to `pop` or `top` from the corresponding `push`.

5 Abstraction and compositionality

Having established the formal definitions of real-time and causal hb-linearisability, we now turn to their contextual guarantees. That is, we return to our questions originally raised in Sections 1 and 3.

Our first theorem and its associated corollary establishes trace abstraction (or contextual refinement) for (real-time and causal) hb-linearisability. That is, if the implementation object under consideration satisfies hb-linearisability with respect to the corresponding abstract object, any (observable) client behaviour when it uses the implementation is a possible behaviour when it uses the abstract specification.

Theorem 14. *Suppose t is a trace of $C[CS]$. If for any $\tau \in \mathsf{execs}(t)$, τ is* hb*-linearisable with respect to AS, then there exists a valid trace s of $C[AS]$ such that $s \sqsubseteq t$.*

The proof of the theorem amounts to showing that assuming object calls of t are hb-linearisable with respect to an abstract history h then, there is a valid trace t' that is a permutation of t such that: all calls in t' are atomic and in the order given by h.

Corollary 15. *If CS is an* hb*-linearisable implementation of AS, then $AS \sqsubseteq CS$.*

Following Herlihy and Wing, we say hb-linearisability is compositional if two objects that individually satisfy hb-linearisability together satisfy hb-linearisability. For simplicity, we define (and verify) the compositionality property for two objects. This trivially generalises to a composition result for n objects. For causal hb -linearisability, recall the notion of *commutative* specification and *unobtrusive* client from Section 3.2.

Theorem 16. *Let $AS = (\mathbb{I} \cup \mathbb{R}, H)$ be a specification. Suppose $\mathbb{I} = \mathbb{I}_1 \uplus \mathbb{I}_2$ and $\mathbb{R} = \mathbb{R}_1 \uplus \mathbb{R}_2$, such that for each $(h, \mathsf{so}) \in H$, $((\mathbb{R}_2 \times \mathbb{I}_1) \cap \mathsf{so}) = \emptyset \wedge ((\mathbb{R}_1 \times \mathbb{I}_2) \cap \mathsf{so}) = \emptyset$. For $i \in \{1, 2\}$, let $\mathbb{M}_i = \mathbb{I}_i \cup \mathbb{R}_i$ and $AS_i = (\mathbb{M}_i, H|\mathbb{M}_i)$ be the specification restricted to $\mathbb{M}_i$, where $H|\mathbb{M}_i = \{(h|\mathbb{M}_i, \mathsf{so}|\mathbb{M}_i) \mid (h, \mathsf{so}) \in H\}$.*

Consider a trace t of $C[CS_1, CS_2]$ and $\tau \in \mathsf{execs}(t)$.

- *If τ is real-time* hb*-linearisable with respect to each AS_i, then τ is real-time* hb*-linearisable with respect to AS.*
- *If τ is causal* hb*-linearisable with respect to each AS_i and both AS_1 and AS_2 are* commutative*, then τ is causal* hb*-linearisable with respect to AS.*
- *If τ is causal* hb*-linearisable with respect to each AS_i and C is* unobtrusive*, then τ is causal* hb*-linearisable with respect to AS.*

The assumption $((\mathbb{R}_2 \times \mathbb{I}_1) \cap \mathsf{so}) = \emptyset \wedge ((\mathbb{R}_1 \times \mathbb{I}_2) \cap \mathsf{so}) = \emptyset$ ensures that AS can be projected onto two independent objects. It is possible to generalise this by assuming that clients ensure any "cross-object" happens-before requirements in AS. However, such a theorem is more complicated to state formally and hence has been omitted for space reasons.

The consequences of compositionality for real-time hb-linearisability, are similar to those that Herlihy and Wing observed for standard linearisability: we need only add that no so-order is lost when we combine independent suborders.

The results for causal hb-linearisability are less familiar. Consider the commutative specification of a bag, where each `remove` happens-after the corresponding `add`; the second clause of Theorem 16 establishes that two disjoint causal hb-linearisable bags may be combined to produce a "larger" hb-linearisable bag. Next, consider a double queue, as in section 3.2; if we can partition the client into independent, non-synchronising thread groups, such that each group only interacts with a single queue, then the third clause of the theorem tells us that executions of the client with the double queue will be causal hb-linearisable.

6 Conclusion

This paper has developed two modified notions of linearisability for weak memory models based on partially ordered notions of execution. These address the inability of standard linearisability [19] to ensure that programmer expectations about the happens-before relations are met by the objects used (see Section 3). Our work has been integrated with the Alglave et al's (AMT) memory model axioms [3], permitting it to uniformly address a variety of memory models. We enhance the axioms of AMT to address abstract objects, invocation/response events of concrete implementations and the consequent modelling of additional happens-before order for both abstract and implementation levels.

We provide two alternative definitions. Our first extension, real-time hb-linearisability, simply adds an additional condition by requiring that the happens-before requirement of the abstract specification is appropriately reflected by the implementation. The second, causal hb-linearisability, additionally replaces the real-time order preservation property by a happens-before order preservation property. We establish composition and abstraction for our two definitions of linearisability.

The results in this paper advance the state of the art in the following ways: firstly, we obtain a contextual refinement theorem and a composition theorem. Secondly, we build on the framework for memory models created by Alglave et al [3]; so our approach is generic, and applicable to any weak memory model that is encompassed by [3], so we are also able to address TSO, C11 release-acquire, ARM and POWER. Thirdly, our framework permits us to explore both global-time and partial-order time variants of the definition of linearisability.

Related work. Since its introduction by Herlihy and Wing [19], linearisability has emerged as the de-facto criterion for correctness of concurrent objects. We refer

the reader to our survey article [13] for a detailed overview and bibliography of the large amount of research into the verification of linearisability. These investigations were carried out in the context of sequential consistency. Below, we discuss the most closely related papers that explore linearisability in the context of relaxed memory.

The study of linearisability, in the presence of relaxed memory was initiated by Burckhardt et al. [7]. This study was carried out for the TSO memory model. The key idea behind this paper is the association of a separate notion of atomic update of memory to a method call in addition to the usual notion of an atomic execution of the method. Thus, a methods is not atomic in this perspective. In our presentation, the happens before relation in the specification describes the requirements of memory visibility on the implementation. In order to prove hb-linearizability, these requirements have to be established, though we have seen that in some cases this proof is immediate.

A notion of linearisability based on transforming sc histories by delaying returns to an associated flush event has also been explored [14, 10], allowing abstractions to remain atomic. Separately, working in the TSO memory model, Doherty and Derrick [12] study a weakening of linearisability using commutations allowed by the specification. In [9], motivated in part by hardware architectures such as Power and ARM, Derrick and Smith provide a framework for defining linearisability in relaxed memory models by allowing the observable order of the execution to be weaker (and hence different) from the full program order. The causal hb-linearisability definition of our paper follows the intuitions of [9]. In this paper, we identify the observable order in the context of AMT models. Since the AMT models provide a rich framework of relations to describe architectures, our methods apply to a wide class of architectures, thus including TSO, Power, and ARM. Our results also apply to the recent ARM8 proposal [24, 8] by identifying the "observed-before" order of that formalisation as the causal order.

Contextual refinement for the C11 memory model is studied by Batty et al. [4]. They consider histories of events constructed using *guarantees* and *deny* relations [11] — guarantees describe *happens-before* representing synchronisations internal to a library, whereas denies describe orders that cannot be enforced by a client due to the internal synchronisations within a library.

The use of happens-before in specifications to aid abstraction based reasoning has appeared in our prior paper [20]. We provided an order theoretic enhancement of linearisability that addresses TSO, PSO as well as JMM.

In this paper, we have been inspired by a simplified version of [4] (as specialised to handle the release-acquire atomics of C11) and the methods in our own prior paper [20]. In common with [20, 4, 10] and in contrast to [7], our definitions maintain the classical atomic and instantaneous view of method executions in linearisability. In common with all the above papers, we prove abstraction results. In common with [4], but in contrast with [7], we also prove composition results.

Acknowledgements. We thank our anonymous reviewers, whose comments have helped improve this paper.

References

1. Adve, S.V., Boehm, H.J.: Memory models: a case for rethinking parallel languages and hardware. Commun. ACM 53, 90–101 (2010)
2. Adve, S.V., Gharachorloo, K.: Shared memory consistency models: A tutorial. Computer 29(12), 66–76 (1996)
3. Alglave, J., Maranget, L., Tautschnig, M.: Herding cats: Modelling, simulation, testing, and data mining for weak memory. ACM Trans. Program. Lang. Syst. 36(2), 7:1–7:74 (2014)
4. Batty, M., Dodds, M., Gotsman, A.: Library abstraction for C/C++ concurrency. In: POPL. pp. 235–248. ACM (2013)
5. Batty, M., Owens, S., Sarkar, S., Sewell, P., Weber, T.: Mathematizing C++ concurrency. In: POPL. pp. 55–66. ACM (2011)
6. Boehm, H.J., Adve, S.V.: Foundations of the C++ concurrency memory model. In: Gupta, R., Amarasinghe, S.P. (eds.) PLDI. pp. 68–78. ACM (2008)
7. Burckhardt, S., Gotsman, A., Musuvathi, M., Yang, H.: Concurrent library correctness on the TSO memory model. In: Seidl, H. (ed.) ESOP. LNCS, vol. 7211, pp. 87–107. Springer (2012)
8. Deacon, W.: Arm64 cat file. `https://github.com/herd/herdtools7/commit/daa126680b6ecba97ba47b3e05bbaa51a89f27b7` (2017)
9. Derrick, J., Smith, G.: An observational approach to defining linearizability on weak memory models. In: FORTE. pp. 108–123 (2017)
10. Derrick, J., Smith, G., Dongol, B.: Verifying linearizability on TSO architectures. In: IFM. LNCS, vol. 8739, pp. 341–356. Springer (2014)
11. Dodds, M., Feng, X., Parkinson, M.J., Vafeiadis, V.: Deny-guarantee reasoning. In: Castagna, G. (ed.) ESOP. LNCS, vol. 5502, pp. 363–377. Springer (2009)
12. Doherty, S., Derrick, J.: Linearizability and causality. In: SEFM. LNCS, vol. 9763, pp. 45–60. Springer (2016)
13. Dongol, B., Derrick, J.: Verifying linearisability: A comparative survey. ACM Comput. Surv. 48(2), 19 (2015)
14. Dongol, B., Derrick, J., Smith, G., Groves, L.: Defining correctness conditions for concurrent objects in multicore architectures. In: Boyland, J.T. (ed.) ECOOP. LIPIcs, vol. 37, pp. 470–494. Dagstuhl (2015)
15. Dongol, B., Groves, L.: Contextual trace refinement for concurrent objects: Safety and progress. In: Ogata, K., Lawford, M., Liu, S. (eds.) ICFEM. LNCS, vol. 10009, pp. 261–278 (2016)
16. Filipović, I., O'Hearn, P.W., Rinetzky, N., Yang, H.: Abstraction for concurrent objects. Theor. Comput. Sci. 411(51-52), 4379–4398 (2010)
17. Gotsman, A., Yang, H.: Liveness-preserving atomicity abstraction. In: Aceto, L., Henzinger, M., Sgall, J. (eds.) ICALP(2). LNCS, vol. 6756, pp. 453–465 (2011)
18. Herlihy, M., Shavit, N.: The Art of Multiprocessor Programming. Morg. Kauf. (2008)
19. Herlihy, M.P., Wing, J.M.: Linearizability: a correctness condition for concurrent objects. ACM Trans. Program. Lang. Syst. 12(3), 463–492 (1990)
20. Jagadeesan, R., Petri, G., Pitcher, C., Riely, J.: Quarantining weakness - compositional reasoning under relaxed memory models (extended abstract). In: Felleisen, M., Gardner, P. (eds.) ESOP. LNCS, vol. 7792, pp. 492–511. Springer (2013)
21. Lamport, L.: How to make a correct multiprocess program execute correctly on a multiprocessor. IEEE Trans. Computers 46(7), 779–782 (1979)

22. Liang, H., Hoffmann, J., Feng, X., Shao, Z.: Characterizing progress properties of concurrent objects via contextual refinements. In: D'Argenio, P.R., Melgratti, H.C. (eds.) CONCUR. LNCS, vol. 8052, pp. 227–241. Springer (2013)
23. Manson, J., Pugh, W., Adve, S.V.: The Java memory model. In: POPL '05. pp. 378–391 (2005)
24. Pulte, C., Flur, S., Deacon, W., French, J., Sarkar, S., Sewell, P.: Simplifying arm concurrency: Multicopy-atomic axiomatic and operational models for armv8. In: POPL (2018), to appear
25. Sarkar, S., Sewell, P., Alglave, J., Maranget, L., Williams, D.: Understanding power multiprocessors. In: PLDI. pp. 175–186. ACM (2011)
26. Sevcík, J.: Program Transformations in Weak Memory Models. PhD thesis, Laboratory for Foundations of Computer Science, University of Edinburgh (2008)
27. Sewell, P., Sarkar, S., Owens, S., Nardelli, F.Z., Myreen, M.O.: x86-TSO: A rigorous and usable programmer's model for x86 multiprocessors. Commun. ACM 53(7), 89–97 (2010)
28. Shavit, N.: Data structures in the multicore age. Commun. ACM 54(3), 76–84 (2011)
29. Smith, G., Winter, K.: Relating trace refinement and linearizability. Formal Aspects of Computing pp. 1–16 (2017)