Linearizability on hardware weak memory models

Graeme Smith

Kirsten Winter

Robert J. Colvin

School of Information Technology and Electrical Engineering,
The University of Queensland, Australia

Abstract. Linearizability is a widely accepted notion of correctness for concurrent objects. Recent research has investigated redefining linearizability for particular hardware weak memory models, in particular for TSO. In this paper, we provide an overview of this research and show that such redefinitions of linearizability are not required: under an interpretation of specification behaviour which abstracts from weak memory effects, the standard definition of linearizability is sound and complete on *all* hardware weak memory models. We prove our result with respect to a definition of object refinement which takes a weak memory model as a parameter. The main consequence of our findings is that we can leverage the range of existing techniques and tools for standard linearizability when verifying concurrent objects running on hardware weak memory models.

Keywords: Linearizability; Correctness; Concurrent objects

1. Introduction

Linearizability [HW90] is a widely accepted notion of correctness for concurrent objects [MS04] that relates the behaviours of an object's implementation to the possible behaviours of its specification. As a correctness notion it benefits greatly from being *compositional*, i.e., the linearizability of each object of a system in isolation guarantees that the overall system is also linearizable. This provides us with a practical approach to proving correctness.

At the level of implementation, operations on an object take time and hence they may overlap in a multi-threaded program. This is obviously difficult to reason about. Linearizability allows us to prove, however, that the behaviour of such an object implementation is consistent with that of a specification in which operations are atomic, and hence cannot overlap. The key concept is the notion of a *linearization point*, a point where an operation in the implementation can be thought of as taking effect atomically. Choosing
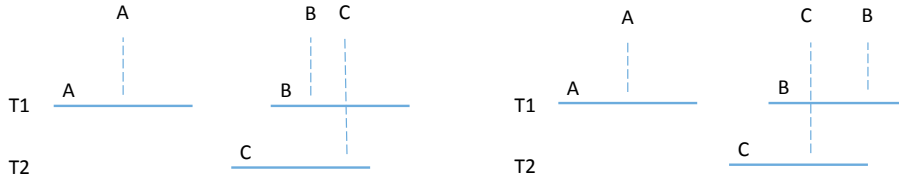
Fig. 1. Linearizability example

such a point for each operation in a concurrent history of an implementation allows us to match that history with a sequential history of the specification. The sequential history is referred to as a *linearization* of the concurrent history.

Figure 1 shows two linearizations of a concurrent history in which operation B of thread T1 overlaps with operation C of thread T2. There are two important things to note. Firstly, the linearization point of an operation must occur somewhere between its *invocation*, i.e., when it is called, and its *response*, i.e., when it returns. This means that operations which do not overlap in the concurrent history occur in the linearization in the order they were invoked. Secondly, overlapping operations in the concurrent history may occur in either order in the linearization, regardless of the order of their invocations and responses. A concurrent history satisfies a specification, provided one of its possible linearizations is a history of the specification.

A concurrent object is considered correct when each of its *finite* histories linearizes with a sequential history of the specification. Hence, linearizability only checks safety properties of an object implementation, not liveness properties [GY11, SW17]. Since linearizability is compositional, we can prove the correctness of a system of interacting objects by showing each component object is linearizable with respect to its specification [HW90].

Recent work [BGMY12, GMY12, TMW13, DSD14, DD16, DJRA18, DDWD18] examines the applicability of linearizability in the context of weak memory models of modern multicore architectures [SSO$^+$10, SSA$^+$11, MHMS$^+$12, AFI$^+$08, AMT14, FGP$^+$16, CS18]. These memory models improve hardware efficiency by reducing accesses to global memory. Individual threads may operate on local copies of global variables, updates to the global memory being made by the hardware and largely out of the programmer's control.[1] This can cause threads executing on different cores to get out of sync with respect to the values of global variables.

For example, on the TSO (Total Store Order) architecture [SSO$^+$10] a thread updating a global variable x stores the new value in a per-core FIFO buffer. Threads executing on that core will then read x from the buffer, rather than the global memory, until the new value is flushed from the buffer by the hardware. In the meantime, threads on other cores read the value of x from the global memory or from their own core's buffer when it has a value for x.

There have been several attempts at defining linearizability for TSO. Burckhardt et al. [BGMY12] include a notion of buffers in the specification of a concurrent object, and associate two atomic steps with each specification operation: one where the effect of the operation updates the buffer, and a subsequent one where it takes effect in the global memory. Gotsman et al. [GMY12] introduce nondeterminism into the specification to model that a thread may, or may not, have seen a recent update. Both of these approaches change the specification that the implementation needs to satisfy. The resulting specifications are less intuitive and do not correspond to specifications that would normally be found as part of a software library.

Derrick et al. [DSD14] take a different approach, leaving the specification unchanged and instead changing the definition of linearizability. A similar approach is used by Travkin et al. [TMW13, TW14] when developing tool support for proving linearizability on TSO. These approaches (which we will collectively call *TSO-linearizability*) change the bounds within which the linearization point must occur. Specifically, they do not require that the linearization point of an operation occurs before its response; it can occur any time up to the final flush associated with the operation. This could allow operation A of Figure 1 to linearize after C in the case when its flush occurs after C (see Figure 2).

TSO-linearizability is the basis for the proof methods in [DSGD14] and [DSGD17]. As proved in [DS15]

---

[1] A programmer can add fences (or memory barriers) to code to force any pending updates to be written to memory. However, if used indiscriminately, fences reduce the gains in efficiency that a weak memory model provides.
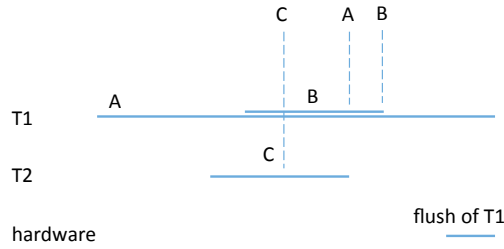
Fig. 2. Linearizability example on TSO

Fig. 3. Program that can result in $z = w = 0$ on TSO

it is compositional. It takes a particular view on the meaning of a specification in which, although operations occur atomically, they do not necessarily take effect immediately. Assume, for example, we have an object o with operations $WriteX$ which writes value 1 to a variable x, $WriteY$ which writes value 1 to a variable y, and $ReadX$ and $ReadY$ which read the variables x and y respectively. When a straightforward implementation of o's operations (i.e., one not involving fences) is run on TSO, the program in Figure 3 could result in both z and w being equal to 0 (the initial value of both variables). This occurs when the flushes of both writes are delayed until after both reads have executed. This somewhat surprising behaviour is considered correct by TSO-linearizability since there is a linearization as shown in Figure 4.

While this is a valid view of correctness, it requires the user to have a solid understanding of the memory model in order to know what behaviour the object might engage in. In particular, as can be seen in Figure 4, operations may effectively occur out-of-order with respect to *program order*, i.e., the order they appear in the program text. Under simple memory models like TSO, this is not too much of a burden for the user, but may become so under more complex memory models, especially when the code of the object also becomes more complex.

Strengthening the definition of TSO-linearizability to ensure that events take effect in program order is proposed in [DS15] and [DDGS15]. Under these definitions, the program in Figure 3 should never result in both z and w equal to 0 if the operations behave according to their specification. While this overcomes the above problem, it introduces a different one: the definition is no longer compositional. To see this, consider the example in Figure 5 where $o1$ and $o2$ are different objects with an operation $Write$ (to write a local variable of the object) and an operation $Read$ (to read the local variable).

Assume the implementation of each object (when run in isolation) is linearizable. When the objects are run together, compositionality would ensure that the combined system is also linearizable. However, the composed system of Figure 5 can result in both x and y being set to 0 on TSO. This is possible since $T2$ may perform its write to $o2$ and read from $o1$ before $T1$ writes to $o1$ (thus setting x to 0), but not flush the value it wrote to $o2$ until after the other two threads have run to completion (setting y to 0).

This outcome is not possible, however, according to a specification based on operations taking effect in
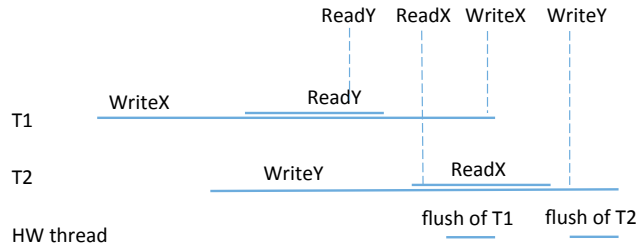

Fig. 4. Linearization of the program of Figure 3

T1		T2		T3

T1

o1.Write(1);   ‖   o2.Write(1);   ‖   while(z=0) z := o1.Read( );
                     x: = o1.Read( )   y := o2.Read( )

Fig. 5. Compositionality counter-example (triangular race)

program order: if x is set to 0 then this step and hence the write to $o2$ on $T2$ must have taken effect before the write to $o1$ on $T1$ took effect, and hence before the read of $o2$ on $T3$. Hence, the composed system is not linearizable.

Doherty and Derrick [DD16] provide a variant of TSO-linearizability which preserves program order and is proved compositional when the client program is restricted to be free from *operation races* (like data races but at the level of operations rather than individual lines of code). The program in Figure 5 provides an example of such a racy program showing a *triangular race* [Owe10] and hence Doherty and Derrick's approach is not applicable. While their result is a useful contribution, it does not allow us to prove linearizability of objects which may be used in *any* context: something we would like for objects in a software library.

What is central to the quest to defining a suitable notion of linearizability is a reference point for correctness under weak memory models that allows one to prove soundness and completeness. What does it mean for an object implementation to behave correctly, and which behaviour of an object can be deemed incorrect? The answer to this seemingly simple question requires some attention in the context of weak memory models. In [SWC18] we provided such a notion of correctness, *object refinement*, which is based on a notion of trace refinement where the object is viewed in the context of a client program. This notion is similar to contextual or observational refinement [DG16, FORY10] but is geared for the context of weak memory models. Object refinement is parameterised by the memory model it refers to, and is therefore generic and can be instantiated for any hardware weak memory model. In this work we utilise this notion to provide a formal proof of soundness and completeness. In contrast to all previous work, we prove that using standard linearizability (as originally proposed by Herlihy and Wing [HW90]) is sound and complete not just on TSO, but *any* hardware weak memory model. It is, of course, also compositional (as proved by Herlihy and Wing).

To do this, we adopt an alternative view of a specification in which its operations occur *and take effect* atomically (rather than potentially taking effect later). This view of specifications has two advantages. Firstly, it ensures that an implementation maintains the *intent* of the specification, e.g., an implementation of a lock operation does not have a delayed effect that allows multiple threads to acquire the associated lock [DSD14]. Secondly, understanding a linearizable object's specified behaviour within a program does not require a knowledge of the memory model. Analogously to a specification of a concurrent object abstracting from the effects of interleaving, the specification in this view also abstracts from the effects of weak memory models.

The paper is structured as follows. In Section 2 we outline the basic concepts of our theory including that of *effects*[2] which is key to our definitions. Based on these concepts, Section 3 formalises the semantics of programs under a given memory model and postulates some basic axioms that we assume of program behaviour under weak memory. The semantics of a concurrent object in the context of a client program, under a memory model, is elaborated in Section 4, distinguishing cases for the specification, in which operations are atomic, and the implementation which includes non-atomic, and possibly non-terminating, operations. Section 5 ties these basics into the notion of *weak-memory trace refinement* which defines refinement under memory model M for a client program using an object and its specification, respectively. The definition is parameterised with a given memory model for which we assume a semantics is given. Using weak-memory trace refinement we then define our notion of *object refinement* under memory model M which delivers the notion of correctness for objects. In Section 6, we prove that the standard definition of linearizability is both sound and complete with respect to our definition of object refinement. Section 7 illustrates how object refinement, and hence standard linearizability, can be used to prove correctness of typical concurrent objects. The paper concludes with a discussion of related work in Section 8.

Contributions. This paper is an extension of our previous paper [SWC18] in which our notion of object refinement was first defined. That paper provided a reference point from which definitions of linearizability

---

[2] Referred to as *observations* in our earlier work [SWC18].

on different hardware weak memory models could be proved sound and complete. The definition of object refinement abstracts from the semantics of particular memory models, instead relying on an operational semantics, such as those for TSO [SSO$^+$10] and the significantly weaker memory models POWER and ARM [FGP$^+$16, CS18], to provide possible program behaviours. As it explicitly avoids out-of-thin-air results, it is not applicable to software weak memory models such as C11 [BOS$^+$11, NMS16].

In this paper, we use that parameterised definition to show that standard linearizability is sound and complete for *all* hardware weak memory models. The assumptions that are required for the proofs are made explicit as axioms of the program semantics. These axioms are derivable from concepts, like object encapsulation and atomicity, and are independent of weak memory model behaviour. This is in contrast to previous work which has changed the definition of linearizability, often for a specific memory model. Earlier in this section, we have provided an overview of this body of earlier work and related the different definitions.

The main consequence of our findings is that we can leverage the range of existing techniques and tools for standard linearizability when verifying concurrent objects running on hardware weak memory models.

2. Client programs and effect events

To investigate the behaviour of concurrent objects under hardware weak memory models, and relate their implementations to their specifications using refinement, we need to consider the calling context. Programs calling the operations of a concurrent object are referred to as *client programs*, or clients for short. A client program P is concurrent, running multiple threads T_i possibly on multiple cores, and is affected by the memory model of the architecture it is running on. For some finite n, we have[3]

$$P \mathrel{\widehat{=}} T_1 \parallel T_2 \parallel \ldots \parallel T_n.$$

Following other work on concurrent objects [HW90, FORY10, DG16], the behaviour of a program is described in terms of *events* that occur. We allow events to be *program steps*, *operation events* or, as introduced in Section 2.1, *effect events*.

Program steps are any steps performed by the client other than calling an object operation. These are assignments, conditional branch instructions (e.g., if or while statements), other control instructions like various forms of fences, atomic read-modify-write instructions which atomically perform these three steps (e.g., the compare-and-swap instruction CAS), and higher-level instructions which can, in many cases, be defined in terms of assignments and/or conditional branches. For example, a statement await(z=1) could be defined as while($z \neq 1$) {}.

Operation events abstract the behaviour of an operation call by a program. They include the *invocation* of the operation (i.e., when it is called) and the operation's *response* (i.e., when it returns). The operation events carry the operation's input and output values as parameters and thus reflect the operation's externally visible behaviour. The internal behaviour of an object is elided.

2.1. Effect events

Central to our definitions is the notion of an *effect event*. Such an event denotes the point in an execution where a program step or an operation can be deemed to have taken effect and (if observable) has been observed by *all* threads T_i of client P. Effect events are where the results of program steps and output values of operations (if any) become known globally.

Note that an operation's internal steps and their effects are not observable on the program level (due to object encapsulation), and hence we record only one effect event per operation call. These operation effects are not used for object refinement. Object refinement is based on the refinement of the client program and only takes into account observable program steps. However, operation effects are essential when defining the semantics of objects in a context of a client running under a weak memory model, as we will see in Section 4.

Introducing effect events allows us to decouple when events occur in a program and when they are observed by other threads, which might not fall together under hardware weak memory models. Program steps and invocation and response events correspond to the points in an execution where the program counter

---

[3] For simplicity, we do not consider dynamically spawned threads.

T1   T2

x := o.A();   await(z=1);
z := 1; || o.C();
o.B();

Fig. 6. Client program

(pc) is at the address of the corresponding instruction. For an invocation or response, this instruction will
be an update to pc, setting it to where the execution should continue. Hence, for any memory model these
events will remain in the order that they occur in the program text.

The semantics specific to the memory model under consideration determines the possible orderings of
the effect events. Effect events occur only after their matching program event or, in the case of an operation,
after the matching invocation. Furthermore, for any program step or operation that writes to memory, its
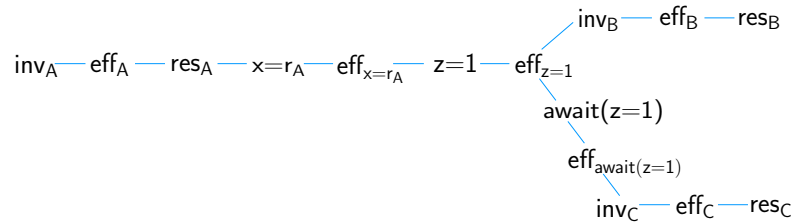effect will occur when *all* threads can either

(a) access the new value of a global program variable written by a program step, or
(b) access the values of *all* shared object variables written by an operation.

2.2. Programs

The behaviour of a concurrent program P under a weak memory model M can be described by a partial
order $<_{P_M}$ over the events of the program to which all possible executions of that program must adhere.
When event e occurs before event e' in the partial order, i.e., $(e, e') \in <_{P_M}$ then event e must occur before
event e' in any execution of P. When P involves branching (e.g., if or while statements), there will be several
paths through P (one corresponding to each combination of branches). The events of different paths need to
be distinct which can be ensured, for example, by including in each event an identifier of the path to which
it belongs.

To understand what a partial order of a program for a particular memory model might look like consider
the example of a client program given in Figure 6 with two global program variables x and z, running on
two threads T1 and T2. The threads call three operations of the same object o, of which the operation B
does not write to any shared object variable. Note that we assume that the result of any operation call is
implicitly stored in a local register (e.g., r_A for the operation call o.A), and inv_A denotes the invocation of
o.A, etc.

On a *sequentially consistent* (SC) architecture, i.e., one without a weak memory model, writes to global
variables occur instantaneously. Hence, the effect event for a program step occurs immediately after the
program step, and that of an operation between its invocation and response. The partial order on the events
of the program of Figure 6 on SC is shown below. The branch in the partial order after the program step
z=1 of thread T1 takes effect corresponds to the operation call o.B() of T1 occurring in parallel with the
code of thread T2.

inv_B — eff_B — res_B

inv_A — eff_A — res_A — $x=r_A$ — $eff_{x=r_A}$ — z=1 — $eff_{z=1}$

await(z=1)

$eff_{await(z=1)}$

inv_C — eff_C — res_C

On TSO [SSO$^+$10], writes to global variables and shared object variables become available to threads
on other cores when they are flushed. These flushes may occur at some later point, but we know that they
occur in the same order as the writes occurred. Hence, in the partial order of the events on TSO (depicted
below), eff_A occurs before $eff_{x=rA}$ which occurs before $eff_{z=1}$. The effect event eff_B can only occur after inv_B
but is not ordered with respect to the other effect events since o.B does not write to any shared variable,
and hence does not have to follow the FIFO order of flushes of writes. Similarly, the effect events $eff_{await(z=1)}$
and eff_C are un-ordered.
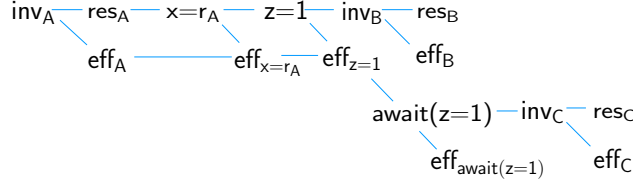
On ARM and POWER [SSA$^+$11, MHMS$^+$12, AFI$^+$08, AMT14, FGP$^+$16, CS18], the effect events might occur out of (program-) order and also flushing of variables need not occur in a FIFO manner. Hence the effects are un-ordered in the partial order of the program depicted below, apart from $\mathsf{eff_A}$ and $\mathsf{eff_{x=r_A}}$ which are ordered due to a data-dependency on register $\mathsf{r_A}$. As under TSO, the assignment to z and the await statement maintain their order due to the data dependency (on z) and the synchronisation (await) between T1 and T2.
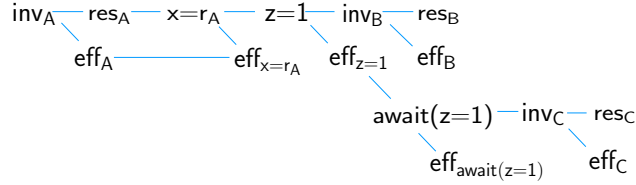
For programs with more than two threads, the semantics of ARM and POWER also allow *non-multi-copy atiomicity*, i.e., where threads on different cores to the thread performing a write may see the write at different times. On such a memory model, the effect will be the point where *all* cores have seen the write. In the operational semantics of ARM and POWER in [CS18], non-multi-copy atomicity is modelled using a sequence of writes where each write is associated with the set of thread identifiers of threads which have seen the write. The effect occurs in this semantics when that sequence indicates that all threads have seen either the write or a *future* write to the same variable, i.e., one that occurs later in the execution.

The order in which effect events occur in an actual behaviour of a program determines the value of the output associated with the effect. This value must also correspond to the output value of the operation response or program step that is observed through the effect event. This correspondence, which prevents out-of-thin air results available in software memory models such as C11 [BOS$^+$11, NMS16], is captured in a well-formedness condition on traces formalised in the next section.

3. Semantics of programs relative to the memory model

The semantics of a program P under memory model M is defined in terms of the set of *events* that can occur, and a partial order $<_{P_M}$ over those events. The events are, on one hand, determined by P through the program text, and, on the other hand, determined by the semantics of the memory model as outlined in Section 2. In this section, we formalise the semantics of a program assuming its events and partial order are given.

3.1. Events

Let T be the set of all thread identifiers, and *Call* the set of all (unique) operation calls. An operation is then defined as a call by a particular thread.

$Op \,\hat{=}\, T \times \textit{Call}$

Let PS denote the set of all program step events, and *Val* the set of all values (of input and output parameters), including a special element $\bot$ meaning 'no value'. The set of all events is defined as follows, where each invocation is associated with an input, and each response and effect with an output.

$Event \,\hat{=}\, step(T, PS) \mid \textit{eff}(T, PS) \mid inv(Op, \textit{Val}) \mid res(Op, \textit{Val}) \mid \textit{eff}(Op, \textit{Val})$

In the remainder of this paper, we refer to $step(T, PS)$ and $\textit{eff}(T, PS)$ as program events, $\mathcal{P}rogEvents$, and to $inv(Op, \textit{Val})$, $res(Op, \textit{Val})$, and $\textit{eff}(Op, \textit{Val})$ as object events, $\mathcal{O}bjEvents$.

A program P has a set of events, $events(P)$, such that for each invocation event in $events(P)$, $events(P)$ also contains all possible corresponding response and effect events. That is, each called operation can respond and take effect and the associated output is not under the program's control.

$$\forall\, op : Op;\; in : Val \bullet$$
$$inv(op, in) \in events(P) \Rightarrow \forall\, out : Val \bullet \{res(op, out), eff(op, out)\} \subseteq events(P) \qquad (1)$$

3.2. Traces

The semantics of a program P is described as a set of *finite* sequences of events, referred to as *traces*.[4] In the following t_i denotes the ith element of a trace t, and $\#t$ its length.

For each trace t, each event is unique (similar events, e.g., calls to the same operation, may be annotated by their relative position in the trace).

$$Trace \,\hat{=}\, \{t : seq\, Event \mid \forall\, i, j \leq \#t \bullet i \neq j \Rightarrow t_i \neq t_j\}$$

The events of a trace and the order on these events are defined as follows.

$$events(t) \,\hat{=}\, \{a : Event \mid \exists\, i \leq \#t \bullet t_i = a\}$$
$$<_t \,\hat{=}\, \{(a, b) : Event \times Event \mid \exists\, i, j \leq \#t \bullet i < j \wedge t_i = a \wedge t_j = b\}$$

Note that the order $<_t$ is a total order over the events in t, as a trace describes exactly one execution of a program.

As a well-formedness condition on traces we postulate that an invocation of an operation always occurs before both the associated response and the associated effect. Similarly, a program step always occurs before its effect. Also, the output value of an operation's effect is the same as that of the corresponding response event. For any trace t we have

$$(\forall\, a : Op;\; out : Val \bullet (\forall\, j \leq \#t \bullet t_j \in \{res(a, out), eff(a, out)\} \Rightarrow \exists\, in : Val;\; i < j \bullet t_i = inv(a, in))) \wedge$$
$$(\forall\, s : T;\; p : PS \bullet (\forall\, j \leq \#t \bullet t_j = eff(s, p) \Rightarrow \exists\, i < j \bullet t_i = step(s, p))) \wedge$$
$$(\forall\, a : Op;\; out_r, out_e : Val \bullet \{res(a, out_r), eff(a, out_e)\} \subseteq events(t) \Rightarrow out_r = out_e) \qquad (2)$$

Since responses remain in the order they appear in the program text, the constraint on the outputs of effects and responses prevents out-of-thin-air results.

3.3. Programs

The semantics of program P on memory model M is defined as the set of traces using only events from P and whose orders adhere to the constraints prescribed by the partial order $<_{P_M}$. That is, if a pair of events (a, b) is in $<_{P_M}$ and b is in an execution of P on M then a occurs earlier in that execution. Hence the semantics of P on M is defined as

$$[\![P]\!]_M \,\hat{=}\, \{t : Trace \mid events(t) \subseteq events(P) \wedge <_{P_M} \Subset <_t\}$$

where $<_{P_M} \Subset <_t$ specifies whether an order is *allowed* by P on M, formally defined as

$$<_{P_M} \Subset <_t \,\hat{=}\, \forall (a, b) :\, <_{P_M} \bullet b \in events(t) \Rightarrow (a, b) \in <_t$$

That is, for any event b that occurs in trace t, if this event is constrained to come after another event a by $<_{P_M}$, then event a must also occur in t before event b. Note that it is not suitable to use the simple subset relation here, i.e., $<_{P_M} \subseteq <_t$, since trace t will not, in general, include all events b that are constrained by $<_{P_M}$.

Since we are aiming at the most general description of program semantics under any memory model, we do not explicitly prescribe $<_{P_M}$. Instead, based on the understanding of concurrent programs, concurrent objects and their interplay (as laid out in Section 2), we formulate certain characteristics of $<_{P_M}$ that are shared by all memory models, and provide a set of axioms in the remainder of this section.

---

[4] Since we are interested in defining a notion of correctness that readily relates to linearizability, and hence only safety properties [GY11, SW17], we do not consider infinite sequences of events in our semantics.

Fig. 7. Scenarios of synchronisations between threads

In the context of this work, we are interested in both the order of operation events and the order between operation and program events in $<_{P_M}$. We define *synchronisation* (between threads) as an event on one thread affecting the occurrence of another event on another thread. Synchronisation requires a writing and a reading access to a shared variable (e.g., through an await statement or a conditional). We make the following assumptions on objects and clients:

- The state space of an object is encapsulated and hence the client does not share any variables directly with the object; communication occurs only through input and output values of the object's operations.
- Consequently, synchronisation between two threads can only occur between program events (referred to as *program synchronisation*) or between operation events (referred to as *object synchronisation*). The client cannot directly synchronise with a step in an operation that occurs between its invocation and response, but only with the invocation and response events themselves which serve as the interface between client and object. Figure 7 depicts the possible synchronisations by means of two scenarios: The scenario on the left illustrates program synchronisation between the threads T1 and T2, in which the program synchronises on two program steps which follow operation A and precede operation B, respectively. An example of this type of synchronisation is given by the instructions z=1 and await(z=1) in Figure 6. The scenario on the right illustrates operations A and B overlapping and synchronising on the object level, outside the control of P.
- A client cannot enforce a flush within an operation; the operation's implementation is outside the client's control.

From these assumptions we can deduce that $<_{P_M}$ can only enforce the invocation of an operation to occur *before* another event, which is not the effect of the operation[5], if and only if the response is also enforced to occur before the event. (The implication from right to left holds due to the well-formedness condition (2).)

$$\forall\, e, inv(a, in), res(a, out) : events(P) \bullet \\ \quad e \neq \mathit{eff}(a, out) \Rightarrow ((inv(a, in), e) \in\, <_{P_M} \Leftrightarrow (res(a, out), e) \in\, <_{P_M}) \tag{3}$$

Similarly, $<_{P_M}$ cannot enforce the response of an operation to come *after* another event, which is not the effect of the operation, unless it also enforces the invocation to come after the event. (As above, the implication from right to left holds due to well-formedness (2).)

$$\forall\, e, inv(a, in), res(a, out) : events(P) \bullet \\ \quad e \neq \mathit{eff}(a, out) \Rightarrow ((e, res(a, out)) \in\, <_{P_M} \Leftrightarrow (e, inv(a, in)) \in\, <_{P_M}) \tag{4}$$

A similar condition holds for the order of effect events with respect to program events. Due to an object's implementation, the effect of an operation might occur before its response event (e.g., if the operation is implemented with fence instructions which prevent a delay in its effect), which is outside the control of P. Hence $<_{P_M}$ cannot enforce the effect of an operation to come *after* another *program* event unless it also enforces the invocation to come after the program event. (Again, the implication from right to left holds due to well-formedness (2).)

$$\forall\, e, inv(a, in), \mathit{eff}(a, out) : events(P) \bullet \\ \quad e \in \mathcal{P}rogEvents \Rightarrow ((e, \mathit{eff}(a, out)) \in\, <_{P_M} \Leftrightarrow (e, inv(a, in)) \in\, <_{P_M}) \tag{5}$$

In some memory models, the order in which two effects can occur can be constrained (e.g., in TSO the order of observable effects follows the order of the observed events), but only if the order of the corresponding

---

[5] The other event can be the response of the operation since partial orders are reflexive and hence $(res(c), res(c))$ is in $<_{P_M}$ for any c.

events is also enforced. Any order that is caused by the nature of a particular implementation (e.g., an additional fence instruction in the code) is outside the control of P (similarly to the case above). This means, if the effects of two operations are ordered then the order of the operations must be enforced, and hence the response of one operation must occur before the invocation of the other.

$$\forall \, inv(a, in), \mathit{eff}(a, out), res(c), \mathit{eff}(c) : events(P) \bullet$$
$$((\mathit{eff}(c), \mathit{eff}(a, out)) \in <_{P_M} \land c \neq (a, out)) \Rightarrow (res(c), inv(a, in)) \in <_{P_M}$$

A similar condition holds if $<_{P_M}$ enforces an operation effect to occur after the invocation, response or effect event of another operation. This is only possible if the two operations are ordered (and hence the response of the first must occur before the invocation of the other). If the operations are not ordered by $<_{P_M}$, they may occur in either order (and hence the effects may occur in either order) or they may overlap. For overlapping operations, $<_{P_M}$ cannot enforce an order of object events since any object synchronisation is beyond the control of P. Therefore we can generalise the above axiom as follows.

$$\forall \, inv(a, in_a), res(a, out_a), \mathit{eff}(a, out_a), inv(b, in_b), \mathit{eff}(b, out_b) : events(P) \bullet$$
$$\forall \, e : \{inv(a, in_a), res(a, out_a), \mathit{eff}(a, out_a)\} \bullet$$
$$((e, \mathit{eff}(b, out_b)) \in <_{P_M} \land a \neq b) \Rightarrow (res(a, out_a), inv(b, in_b)) \in <_{P_M} \qquad (6)$$

As a consequence of these observations we deduce that an order between object events of two operations can only be enforced by $<_{P_M}$ if $<_{P_M}$ also enforces that these two operations do not overlap.

Lemma 1 (Enforced ordering on object events).

$$\forall \, inv(a, in_a), res(a, out_a), \mathit{eff}(a, out_a), inv(b, in_b), res(b, out_b), \mathit{eff}(b, out_b) : events(P) \bullet$$
$$\forall \, e : \{inv(a, in_a), res(a, out_a), \mathit{eff}(a, out_a)\}; \; e' : \{inv(b, in_b), res(b, out_b), \mathit{eff}(b, out_b)\} \bullet$$
$$((e, e') \in <_{P_M} \land a \neq b) \Rightarrow (res(a, out_a), inv(b, in_b)) \in <_{P_M}$$

The proof follows the simple application of Axioms (3) to (6) to all combinations of invocation, response and effect events of the two operations.

4. Semantics of objects under weak memory models

To define object refinement, our notion of correctness, in Section 5, we need to constrain the behaviour of a program to a particular object or collection of objects (called an *object system* in [FORY10]). In either case, the interface between the program and the one or more objects is the operations of those objects, the outputs of which, when the operations are called in a given sequence, reflect the objects' behaviour including any interactions between them. For ease of presentation, we consider a single object only, but the results also hold for a collection of interacting objects whose interface is the operations of all objects in the collection.

Let $t_{|ir}$ denotes the trace t restricted to invocation and response events. The semantics of an object is given as a set of *histories*, where each history is a trace with only invocation and response events.

$$History \mathrel{\hat=} \{t : Trace \mid t_{|ir} = t\}$$

4.1. Object implementation under weak memory models

An object implementation C has a set of object events, $events(C)$, and, on a particular memory model M, a prefix-closed set of histories made up of those events, $[\![C]\!]_M$.[6]

For any object implementation C, $P[C]$ denotes the object C operating in program P. It is only defined when all invocation events of P are events of C. The semantics of C operating in P on memory model M, $[\![P[C]]\!]_M$, is given as those traces of P on M whose object events correspond to a history of C on M.

$$[\![P[C]]\!]_M \mathrel{\hat=} \{t : [\![P]\!]_M \mid \exists \, h : [\![C]\!]_M \bullet t_{|ir} = h\}$$

---

[6] The set of histories of C would be derived in an operational manner based on the weak memory model semantics. This may involve two passes in which firstly effects and their placement are taken into account, and secondly response values are added to reflect this placement, and effects are discarded.

	T1		T2
	lock.acquire;		lock.acquire;
	y := 1	‖	z := 1

Fig. 8. Client program using lock

4.2. Object specification under weak memory models

An object specification A similarly has a set of invocation and response events, $events(A)$, and a prefix-closed set of histories, $[\![A]\!]$ (where prefixes are restricted to complete histories [HW90] in which the final event cannot be an invocation). Since A represents a typical specification found in a software library, its set of histories is *independent* of the memory model (hence there is no subscript). Any weak memory model behaviour is absent from its histories due to its operations being *atomic*, i.e., they respond and take effect immediately and hence occur without interference from other operations. That is, every history in $[\![A]\!]$ is a sequence of invocation and response pairs, i.e., every invocation is followed by its response, and no overlapping of operations is possible.

$$
\begin{aligned}
\forall\, h : [\![A]\!] \bullet \\
(\forall\, i \le \#h \bullet h_i = inv(a, in) &\Rightarrow (i < \#h \land \exists\, out : Val \bullet h_{i+1} = res(a, out))) \land \\
(\forall\, i \le \#h \bullet h_i = res(a, out) &\Rightarrow (i \ne 1 \land \exists\, in : Val \bullet h_{i-1} = inv(a, in)))
\end{aligned} \tag{7}
$$

To capture atomicity of specifications in our semantics, the behaviour of a client using specification A is restricted to those traces where only one operation is active at a time. For example, suppose the specification of a lock object, lock, has an operation acquire which waits until the value of a variable of the object, x, is 1 and sets it to 0, i.e., acquire is specified as await(x=1); x=0. Assuming x is initially 1, in the program of Figure 8 the intention would be that only one of y or z would be set to 1.

On SC, this intention is achieved when the invocation of acquire which occurs second does not happen until after the first acquire has taken effect. On TSO and assuming T1 and T2 are running on different cores, the intention is achieved when the invocation of acquire which occurs second does not happen until after the flush of x from the acquire which occurs first. This flush might be delayed. On ARM and POWER (again assuming T1 and T2 are on different cores), the intention is achieved when the second invocation of acquire does not happen until after the write to x by the first invocation has been seen by all threads (see description of non-multi-copy atomicity in Section 2.1).

In all cases, the intention is met when the second occurrence of acquire is not invoked before the effect event of the first occurrence. In general, for any object specification to behave as intended within a client context, an operation invocation does not occur before the effect event of a previously invoked operation. (Note that due to Axiom (7) invocations also must occur after responses of a previous operation.)

Provided all invocation events of a program P are events of a specification A, $P[A]$ denotes the program P operating with an abstract object whose behaviour satisfies A. To model atomicity, operations can only be invoked after previous operations have taken effect (as motivated above). The semantics of $P[A]$ is given as follows.

$$
\begin{aligned}
[\![P[A]]\!]_M \,\widehat{=}\, \{\ t : [\![P]\!]_M \mid \exists\, h : [\![A]\!] \bullet t_{\,|ir} = h\ \land \\
\forall\, c : Op \times Val;\ k \le \#t \bullet t_k = inv(c) \Rightarrow \\
\forall\, a : Op;\ in : Val;\ i < k \bullet t_i = inv(a, in) \Rightarrow \\
\exists\, out : Val;\ j < k \bullet t_j = \mathit{eff}(a, out)\ \}
\end{aligned}
$$

5. Object refinement

Correctness of an object is defined from the client program's point of view. Such a program can only observe changes to *program variables*, i.e., variables that are not defined locally on a thread or as part on an object. Let $t_{\,|global}$ denote the *observable behaviour* of a trace t, i.e., the sequence of effect events of program steps which write to global program variables. If for two traces t and t' we have $t_{\,|global} = t'_{\,|global}$, we call them *matching* traces.

A program P using C on memory model M refines P using A on M when any observable behaviour of the former is a possible observable behaviour of the latter, i.e., each concrete trace has a matching abstract trace. We refer to this property as *weak-memory trace refinement*.

Definition 1. Weak-memory trace refinement

$$P[A] \sqsubseteq_M P[C] \ \hat{=} \ \forall t : [\![P[C]]\!]_M \bullet \exists t' : [\![P[A]]\!]_M \bullet t'_{|global} = t_{|global}$$

An object implementation C refines an object specification A under weak memory model M if for all possible client programs P, P using C weak-memory trace refines P using A under memory model M. We refer to this property as *object refinement*.

Definition 2. Object Refinement under memory model M

$$A \sqsubseteq_M C \ \hat{=} \ \forall P \bullet P[A] \sqsubseteq_M P[C]$$

If $A \sqsubseteq_M C$ we say that C is correct with respect to A under memory model M.

6. Linearizability

Linearizability relates histories of an object implementation, which may have *pending* invocations, i.e., invocations for which there is no response, to histories of an object specification which do not [HW90]. To do this, it needs to *complete* the implementation histories. This can be done by adding a response when a pending invocation is deemed to have taken effect, and removing the invocation when it has not [HW90]. For example, consider a history comprising a read operation of a variable x occurring on a thread T1 concurrently with a write operation to x on thread T2 where the latter has not yet responded. If the read operation returns the value from the write operation, we can assume the write operation has taken effect and hence we add a response to the history. If the read operation returns the value of x from before the write operation, we can assume the latter has not taken effect and remove its pending invocation.[7]

To define linearizability, functions are put in place for adding responses, and removing invocations from histories. The function *ext* returns the set of traces which extend a given trace with a sequence of responses such that the result is still a trace, i.e., responses are only added for pending invocations.

$$ext(t) \ \hat{=} \ \{t \frown tr : Trace \mid \forall i \leq \#tr \bullet \exists c : Op \times Val \bullet tr_i = res(c)\}$$

The function *comp* returns the trace resulting from the removal of all invocations from a given trace which have neither an effect nor a response.

$$comp(t) \ \hat{=} \ \begin{cases} \langle\rangle, \ \text{if } t = \langle\rangle \\ comp(tail\ t), \ \text{if } NoResp(t) \\ \langle head\ t\rangle \frown comp(tail\ t), \ \text{otherwise} \end{cases}$$

where $NoResp(t) \ \hat{=} \ \exists a : Op;\ in : Val \bullet$
$$head(t) = inv(a, in) \wedge \nexists out : Val;\ i \leq \#t \bullet t_i \in \{res(a, out), eff(a, out)\}$$

The following formalisation of the standard definition of linearizability is based on that of Derrick et al. [DSW11] which has been provedto correspond to the original definition by Herlihy and Wing [HW90]. (Note that $comp(h^+)$ removes all pending invocations from h^+ since histories do not have effect events.)

Definition 3. Linearizability (standard definition)

$$C \ \mathsf{lin}_M \ A \ \hat{=} \ \forall h : [\![C]\!]_M \bullet \exists h' : [\![A]\!] \bullet \exists h^+ : ext(h) \bullet comp(h^+) \sim h' \wedge \prec_{comp(h^+)} \subseteq \prec_{h'}$$

where $t \sim t'$ denotes that t and t' are *thread equivalent*, i.e., when restricted to the events of any one thread they have the same sequence of invocations and responses, and $\prec_t \hat{=} \{(res(c), inv(d)) : <_t\}$, i.e., $\prec_t$ captures the order between operations in a trace (where an operation comes before another if its response is before the other's invocation).

---

[7] In this case, we could also add a response since the write could have taken effect, but after the read took effect. However, to be consistent with the original definition of linearizability, we allow invocations to be removed.

The intuition behind the definition is that operations which are overlapping in $comp(h^+)$ are not ordered by $\prec_{comp(h^+)}$ and, with $\prec_{h'}$ being a superset of $\prec_{comp(h^+)}$, can occur in any order in h'. For example, the overlapping operations B and C of the implementation history of Figure 1 can occur in any order in a linearization of that history. This is equivalent to letting the linearization points of B and C occur anytime between the respective operations' invocations and responses.

Importantly, the definition is *compositional* (this property is proved for the above definition by Herlihy and Wing who refer to it as *locality* [HW90]). In the case when the object implementation C is a collection of interacting objects, compositionality allows us to prove that C is linearizable to a specification of a similar collection of interacting objects by proving each individual object implementation is linearizable to the corresponding object specification.

In the remainder of this section, we prove that standard linearizability is sound and complete with respect to our definition of object refinement. To facilitate the proofs, we introduce the following lemmas on completions of traces whose proofs are included in Appendix A.

Lemma 2. If the events of a trace t are events of a program P then so are the events of any completion of t.

$$\forall P \bullet \forall t : Trace \bullet \forall t^+ : ext(t) \bullet events(t) \subseteq events(P) \Rightarrow events(comp(t^+)) \subseteq events(P)$$

Lemma 3. If a trace t is allowed by a program P on memory model M then so is any completion of trace t that only adds responses for operations whose effects have occurred.

$$\forall P, M \bullet \forall t : Trace \bullet \forall t \frown tr : ext(t) \bullet$$
$$(\forall i \leq \#tr;\ c : Op \times Val \bullet tr_i = res(c) \Rightarrow \exists j \leq \#t \bullet t_j = \mathit{eff}(c)) \wedge <_{P_M} \Subset <_t \Rightarrow <_{P_M} \Subset <_{comp(t \frown tr)}$$

Lemma 4. The operation order of a completion of a trace t is a subset of that of t.

$$\forall t : Trace \bullet \forall t^+ : ext(t) \bullet \prec_{comp(t^+)} \subseteq \prec_t$$

6.1. Soundness

The soundness of standard linearizability in the context of weak memory model M can now be proved in relation to our notion of object refinement under M.

Theorem 1. If an object implementation C linearizes with an object specification A on memory model M then for all client programs P, $P[C]$ is a weak-memory trace refinement of $P[A]$ on M.

$$C \mathsf{\ lin}_M\ A \Rightarrow \forall P \bullet P[A] \sqsubseteq_M P[C]$$

The proof of this theorem takes any concrete history h of C and any abstract history h' of A, where the latter is a linearization of the former, and any trace t of a program calling C that corresponds to the concrete history h. From these ingredients we construct an abstract trace t' such that t' corresponds to the linearization h'. That is, we move the object events of t such that the operations do not overlap and also occur in the same order as in h' in the resulting trace t'. It then remains to show that the trace that is constructed in this way is indeed a trace of the client calling the abstract object (i.e., that t' is a trace of $P[A]$).

Proof
Expanding lin_M and $\sqsubseteq_M$ we have

$$(\forall h : [\![C]\!]_M \bullet \exists h' : [\![A]\!] \bullet \exists h^+ : ext(h) \bullet comp(h^+) \sim h' \wedge \prec_{comp(h^+)} \subseteq \prec_{h'}) \Rightarrow$$
$$\forall P \bullet \forall t : [\![P[C]]\!]_M \bullet \exists t' : [\![P[A]]\!]_M \bullet t'_{|global} = t_{|global}$$

We need to prove that the consequent holds whenever the antecedent does. For any P, either $[\![P[C]]\!]_M = \varnothing$

and the consequent trivially holds, or there exists a trace t in $[\![P[C]]\!]_M$. In the latter case, we have that there exists an h in $[\![C]\!]_M$ such that $t_{|ir} = h$ (from the definition of $[\![P[C]]\!]_M$).

Given such a t and h, when the antecedent holds we also have an $h' \in [\![A]\!]$ and an $h^+ \in ext(h)$ such that

$$comp(h^+) \sim h' \ \wedge \ \prec_{comp(h^+)} \subseteq \prec_{h'} \tag{S1}$$

There may be a number of choices for h^+ based on how many pending invocations are given responses. We choose h^+ so that there is an added response for exactly those pending invocations whose effects occur in t. This is always possible since we know that there exists at least one extension and related abstract history. Call them h_0^+ and h_0', respectively. h_0^+ cannot have less than the required responses. If it did, $comp(h^+)$ would be left with a pending invocation (with an effect) but no response. Hence, $comp(h^+) \sim h'$ would not hold. If h_0^+ has more than the required responses, since $[\![A]\!]$ is prefixed-closed, we can find an h' which is a subsequence of h_0' which does not have the additional operations corresponding to the extra responses.[8] This h' will satisfy (S1) for our chosen $h+$.

Since $t \in [\![P[C]]\!]_M$, we can deduce from the definition of $[\![P[C]]\!]_M$ that $t \in [\![P]\!]_M$, and with the definition of $[\![P]\!]_M$, we have that

$$events(t) \subseteq events(P) \ \wedge \ <_{P_M} \subseteq <_t \tag{S2}$$

Let $t^+ \in ext(t)$ be the trace that extends t with the same sequence of responses that h^+ extends h. It follows that

$$t^+{}_{|ir} = h^+ \ \wedge \ comp(t^+)_{|ir} = comp(h^+) \tag{S3}$$

From (S2), Lemma 2 and Lemma 3 (and since we choose h^+ to only add responses to h for operations whose effects occur in t) we have

$$events(comp(t^+)) \subseteq events(P) \ \wedge \ <_{P_M} \subseteq <_{comp(t^+)} \tag{S4}$$

Since $t^+{}_{|ir} = h^+$ (S3) and $comp(h^+) \sim h'$ (S1) we can deduce that

$$comp(t^+{}_{|ir}) \sim h' \tag{S5}$$

Central to our proof is the fact that we can construct the abstract trace t' appearing in the consequent of the theorem from trace $comp(t^+)$ and history h'. This is done by means of a transformation function $trans$, which reorders object events in $comp(t^+)$ to match the order in h', while maintaining the order of program events. We let $t' = trans(comp(t^+), h', \langle\rangle)$ using the following definition of the transformation function.

Definition 4 (Transformation function). Let $a \in Op$ be an operation, $in, out \in Val$ be inputs and outputs, and $r \in \text{seq} \, \mathcal{O}bjEvents$ a (remainder) sequence that temporarily stores invocations which will be placed into the resulting trace at a later point. The latter allows the invocations in the original trace to be reordered in the resulting trace.
Then $trans(t, h', r)$ is equal to either

$(T1)$ $\quad \langle inv(a, in), eff(a, out), res(a, out)\rangle \frown trans(tail \, t, tail(tail \, h'), r)$
$\qquad\qquad\qquad\qquad$ if $head \, t = head \, h' = inv(a, in) \wedge \exists i \leq \#t \bullet t_i = res(a, out),$

$(T2)$ $\quad trans(tail \, t, h', r \frown \langle inv(a, in)\rangle)$ $\quad$ if $head \, t = inv(a, in) \wedge head \, h' \neq inv(a, in),$

$(T3)$ $\quad trans(tail \, t, h', r)$ $\qquad\qquad\qquad$ if $head \, t \in \{res(a, out), eff(a, out)\},$

$(T4)$ $\quad \langle head \, t\rangle \frown trans(tail \, t, h', r)$ $\qquad$ if $head \, t \in \mathcal{P}rogEvents \wedge \nexists i \leq \#r \bullet r_i = head \, h',$ or

$(T5)$ $\quad \langle inv(a, in), eff(a, out), res(a, out)\rangle \frown trans(t, tail(tail \, h'), r - \{inv(a, in)\})$
$\qquad\qquad\qquad\qquad$ if $\exists i \leq \#r \bullet r_i = head \, h' = inv(a, in) \wedge \exists i \leq \#t \bullet t_i = res(a, out).$

Note that (T1) – (T5) cover all cases when t has a response for each invocation (as $comp(t^+)$ does). If the head of t is an invocation, it is either added to t' (along with the corresponding effect and a matching response) (T1) or added to r (T2). If it is a response or effect, it is removed (T3). If it is a program event, and the head of h' is not in r, it is added to t' (T4). If the head of h' is in r, it is added to t' (along with

---

[8] Since there is at most one pending invocation per thread, such an h' will be in the prefix-closed set $[\![A]\!]$.

the corresponding effect and a matching response) (T5). Since $comp(t^+{}_{|ir}) \sim h'$ (S5), all invocations added to r will eventually be processed via the recursive definition.

It is easy to prove that cases (T1) – (T5) of Definition 4 ensure that the newly constructed trace t' contains the same invocation and response events as $comp(t^+)$ and its restriction to invocation and response events matches h'.

$$events(t'{}_{|ir}) = events(comp(t^+){}_{|ir}) \quad \wedge \quad t'{}_{|ir} = h' \tag{S6}$$

Cases (T1) and (T5) in particular ensure that in the transformed trace the operations do not overlap (as invocations, effects and responses of each operation are added to t' in one step).

$$\begin{aligned}
&\forall\, c : Op \times Val;\ k \leq \#t' \bullet t'_k = inv(c) \Rightarrow \\
&\quad \forall\, a : Op;\ in : Val;\ i < k \bullet t'_i = inv(a, in) \Rightarrow \exists\, out : Val;\ j < k \bullet t'_j = eff(a, out)
\end{aligned} \tag{S7}$$

With case (T4) we also ensure that program steps are in the same order in t and t'.

$$t'{}_{|p} = t{}_{|p} \tag{S8}$$

where $t{}_{|p}$ denotes the trace t restricted to program events. Hence, we have

$$t'{}_{|global} = t{}_{|global} \tag{S9}$$

To prove the consequent of Theorem 1 it remains to show that $t' \in [\![P[A]]\!]_M$. Given the definition of $[\![P[A]]\!]_M$ and (S6) and (S7) this reduces to proving that $t' \in [\![P]\!]_M$, i.e., $t' \in Trace$, $events(t') \subseteq events(P)$ and $<_{P_M} \subseteq <_{t'}$.

i.) $t' \in Trace$:

With $t \in [\![P[C]]\!]_M$ and Lemmas 2 and 3 it follows that $comp(t^+) \in [\![P]\!]_M$ and hence is a *Trace*.

With $events(t'{}_{|ir}) = events(comp(t^+){}_{|ir})$ (S6) and $t'{}_{|p} = t{}_{|p}$ (S8) and the fact that effects are only added to t' to match existing responses ((T1) and (T5)), it follows that all events in t' are unique.

Due to cases (T1) and (T5) it is guaranteed that the order of object events in t' is well-formed, i.e., for all $inv(a, in)$ in t' we have $inv(a, in) <_{t'} res(a, out) \wedge inv(a, in) <_{t'} eff(a, out)$.

From case (T4) it follows that the order of program events (program steps and their effects) is maintained in t', i.e., for all $step(s, p)$ and $eff(s, p)$ in t' we have $step(s, p) <_{t'} eff(s, p)$.

From the above it can be deduced that since t is a well-formed trace, t' is also a well-formed trace, i.e.,

$$t' \in Trace \tag{S10}$$

ii.) $events(t') \subseteq events(P)$:

With $events(comp(t^+)) \subseteq events(P)$ ((S2) and Axiom (1)) and $events(t'{}_{|ir}) = events(comp(t^+){}_{|ir})$ (S6) and $t'{}_{|p} = t{}_{|p}$ (S8) and the fact that effects are only added to t' to match existing responses ((T1) and (T5)) and these are in $events(P)$ due to Axiom (1), it follows that

$$events(t') \subseteq events(P) \tag{S11}$$

iii.) $<_{P_M} \subseteq <_{t'}$:

To prove this property it must be ensured that $\forall (a, b) :\ <_{P_M} \bullet b \in events(t') \Rightarrow (a, b) \in <_{t'}$. This property is split into the following four cases.

a.) The order between two program events that is enforced by $<_{P_M}$ is maintained in t'.

$$\forall\, a, b : \mathcal{P}rogEvents \bullet (a, b) \in <_{P_M} \wedge b \in events(t') \Rightarrow (a, b) \in <_{t'}$$

b.) If $<_{P_M}$ enforces that a program event has to occur before an object event then this order is maintained by t'.

$$\forall\, p : \mathcal{P}rogEvents;\ e \in \mathcal{O}bjEvents \bullet (p, e) \in <_{P_M} \wedge e \in events(t') \Rightarrow (p, e) \in <_{t'}$$

c.) If $<_{P_M}$ enforces that an object event has to occur before a program event then this order is maintained by t'.

$$\forall\, p : \mathcal{P}rogEvents;\ e \in \mathcal{O}bjEvents \bullet (e, p) \in <_{P_M} \wedge p \in events(t') \Rightarrow (e, p) \in <_{t'}$$

d.) The order between two object events that is enforced by $<_{P_M}$ is maintained in t'.

$$\forall\, a, b : \mathcal{O}bjEvents \bullet (a, b) \in <_{P_M} \wedge b \in events(t') \Rightarrow (a, b) \in <_{t'}$$

These four remaining properties on t' rely firstly on assumptions on $<_{P_M}$ relating to constraints on object events that might be enforced by P and M, which were outlined in Section 3, and secondly on properties of *trans* relating to the order of program events in t' which can be deduced from Definition 4. The proofs are provided in Appendix B. □

6.2. Completeness

Theorem 2. If we have an object implementation C and specification A such that, for all programs P, $P[C]$ is a weak-memory trace refinement of $P[A]$ on memory model M then C linearizes to A on M.

$$(\forall\, P \bullet P[A] \sqsubseteq_M P[C]) \;\Rightarrow\; C\ \mathsf{lin}_M\ A$$

The proof of this theorem is based on the notion of a "strongest" client program $\widehat{P}$ for each implementation history h (the details of which are explained below). When such a strongest client program exists, the antecedent of the theorem implies $\widehat{P}[A] \sqsubseteq_M \widehat{P}[C]$ and we use this fact to prove that there exists an $h' \in [\![A]\!]$ such that h linearizes to h'. In the case that such a strongest client program does not exist, we show that either (i) the theorem is trivially true, or (ii) there must be an $h' \in [\![A]\!]$ such that h linearizes to h'. Applying these arguments to all h in $[\![C]\!]_M$ results in the theorem's consequent being true.

The strongest client program $\widehat{P}$ of a history h is one which

1. allows h to occur enforcing the order of its operations, and
2. records every invocation and response in a global variable without delay.

Property 1 requires that $\widehat{P}$ enforces the order of operations (i.e., the response/invocation order) to be as in h. Although $\widehat{P}$ cannot control the outputs of operations, if it calls the operations with the inputs in h the corresponding outputs will be possible (since h is an implementation history). For operations on a particular thread, the required order is enforced by $\widehat{P}$'s program order. For operations on different threads, it can be achieved using inter-thread program synchronisation where necessary. For example, if the implementation history h refers to two threads, n and m, and the response of $OpA(x)$ on n occurs before the invocation of $OpB(y)$ on m in h then $\widehat{P}$ could be of the form

...; $u := OpA(x)$; $z := 1$; ... $||$ ...; $await(z = 1)$; $v := OpB(y)$; ...

where initially $z = 0$ and the elided parts of the program do not change z.

Property 2 can be achieved, for example, with $\widehat{P}$ having global variables var_{inv} and var_{res} and every operation call $OpA(x)$ on thread n which returns a value appearing in $\widehat{P}$ as $var_{inv} = (n, OpA, x)$; *fence*; $y = OpA(x)$; $var_{res} = (n, OpA, y)$; *fence*, and every operation call $OpA(x)$ which doesn't return a value as $var_{inv} = (n, OpA, x)$; *fence*; $OpA(x)$; $var_{res} = (n, OpA, \perp)$; *fence*. Note that the recording of invocations and responses using var_{inv} and var_{res} is decoupled from the actual operation invocations and responses, and hence interleaving of the recording events from different threads is possible.

It is not necessarily the case that matching traces of such a strongest client program have the same sequence of invocations and responses. This is due to the fact that an invocation might be recorded (via var_{inv}) but the operation not (yet) called, or an invocation and response have occurred but the response is not yet recorded (recall that traces are prefix-closed). Nevertheless, for each abstract trace that matches a concrete trace we can always find another abstract trace that also matches the concrete trace and additionally has the same sequence of invocations and responses as the completed concrete trace.

Lemma 5. For an object implementation C which is an object refinement of a specification A under M, and any client program P that records every invocation and response of an operation without delay, we have

$$\forall\, t : [\![P[C]]\!]_M \bullet (\exists\, t' : [\![P[A]]\!]_M \bullet t_{|global} = t'_{|global}) \Rightarrow$$
$$\exists\, t'' : [\![P[A]]\!]_M \bullet t_{|global} = t''_{|global} \wedge \exists\, t^+ : ext(t) \bullet comp(t^+)_{|ir} \sim t''_{|ir}$$

The proof of this lemma is based on the prefix closure of traces and can be found in Appendix C.

Proof of Theorem 2

Expanding lin_M and $\sqsubseteq_M$ we have

$$(\forall P \bullet \forall t : [\![P[C]]\!]_M \bullet \exists t' : [\![P[A]]\!]_M \bullet t'_{\,|global} = t_{\,|global}) \Rightarrow$$
$$\forall h : [\![C]\!]_M \bullet \exists h' : [\![A]\!] \bullet \exists h^+ : ext(h) \bullet comp(h^+) \sim h' \wedge \prec_{comp(h^+)} \subseteq \prec_{h'}$$

Assuming the antecedent is true, we prove the consequent for any $h \in [\![C]\!]_M$. It will not be possible to construct a strongest client program for h when the synchronisation between two threads, added explicitly or via a fence, changes the behaviour of the operation occurring after the synchronisation. In the example above, if synchronising using z causes $OpB(y)$ to result in a different output then the resulting program will not allow h to occur. This situation would arise, for example, on TSO where the effect of $OpA(x)$ would necessarily occur before the effect of $z = 1$ due to the FIFO order of the per-core buffer. In other words, there will be no strongest client program when the behaviour of one operation ($OpB(y)$ in the example) relies on the effect of an earlier operation ($OpA(x)$ in the example) being delayed.

For such a h, there are two possibilities.

(i) We can construct a program P such that $P[C]$ has an observable behaviour not in $P[A]$ due to the delayed effect in h (since a similar delayed effect is not possible in $P[A]$), i.e., $\exists P \bullet \exists t : [\![P[C]]\!]_M \bullet t_{\,|ir} = h \wedge \forall t' : [\![P[A]]\!]_M \bullet t'_{\,|global} \neq t_{\,|global}$. In this case, the antecedent of the theorem does not hold, and the theorem is trivially true.

(ii) We cannot construct such a program P and hence $\forall P \bullet \forall t : [\![P[C]]\!]_M \bullet t_{\,|ir} = h \Rightarrow \exists t' : [\![P[A]]\!]_M \bullet t'_{\,|global} = t_{\,|global}$. In this case, consider a program P_0 which allows the history h to occur, and indicates (for any other history) when the operations occur in the same order as in h by means of triangular races. For the example with OpA and OpB, P_0 is of the form

$$z = 1; \;||\; ...; \; u = OpA(x); \; w = z; \; ... \;||\; ...; \; await(z = 1); \; v = OpB(y); \; ...$$

where z is initially 0 and the elided parts of the program do not change z, w or v. If P_0 results in $w = 0$, we can deduce that $OpA(x)$ has occurred before $OpB(y)$. Assume $OpB(y)$ in C returns 1 when $OpA(x)$'s effect has taken place and 0 otherwise. For history h, the program $P_0[C]$ will additionally result in $v = 0$ since the effect of $OpA(x)$ is delayed. Since we know there is an abstract trace t' matching each concrete trace t where $t_{\,|ir} = h$, it must be possible to have the result $w = v = 0$ from $P_0[A]$. Therefore, there must be a history $h' \in [\![A]\!]$ in which $OpA(x)$ occurs before $OpB(y)$ and the latter operation outputs 0 (even though the effect of $OpA(x)$ is not delayed). This history h' is a linearization of h. Hence the consequent of the theorem holds for h.

For all other $h \in [\![C]\!]_M$, a strongest program $\widehat{P}$ can be constructed. From Property 1, we know that $\widehat{P}$ allows h to occur, i.e., there exists a trace of $\widehat{P}[C]$ corresponding to h, and that $\widehat{P}$ enforces the operations to occur in the order in h.

$$\exists t : [\![\widehat{P}[C]]\!]_M \bullet t_{\,|ir} = h \tag{C1}$$

$$\forall c, d : Op \times Val \bullet (res(c), inv(d)) \in <_h \Rightarrow (res(c), inv(d)) \in <_{\widehat{P}_M} \tag{C2}$$

With Lemma 5 and the antecedent of the theorem we can deduce that if a concrete trace has a matching abstract trace there always exists another matching trace which has the same invocations and responses.

$$\forall t : [\![\widehat{P}[C]]\!]_M \bullet \exists t'' : [\![\widehat{P}[A]]\!]_M \bullet t_{\,|global} = t''_{\,|global} \wedge \exists t^+ : ext(t) \bullet comp(t^+)_{\,|ir} \sim t''_{\,|ir} \tag{C3}$$

Let $t \in [\![\widehat{P}[C]]\!]_M$ be any trace satisfying (C1), i.e.,

$$t_{\,|ir} = h \tag{C4}$$

There exists an abstract trace $t'' \in [\![\widehat{P}[A]]\!]_M$ and a trace $t^+ \in ext(t)$ such that the completion of t^+ and the abstract trace share the same invocations and responses (from (C3)).

$$comp(t^+)_{\,|ir} \sim t''_{\,|ir} \tag{C5}$$

With Lemma 4, it follows that

$$\prec_{comp(t^+)} \subseteq \prec_t \tag{C6}$$

Since $t'' \in [\![\widehat{P}[A]]\!]_M$, from definition of $[\![\widehat{P}[A]]\!]_M$ we have that $t'' \in [\![\widehat{P}]\!]_M$ and hence

$$events(t'') \subseteq events(\widehat{P}) \ \wedge \ <_{\widehat{P}_M} \Subset <_{t''} \ . \tag{C7}$$

and that there exists an $h' \in [\![A]\!]$ such that

$$h' = t''_{|ir} \tag{C8}$$

From (C2) and (C4) we have

$$\forall \, c, d : Op \times Val \bullet (res(c), inv(d)) \in <_t \Rightarrow (res(c), inv(d)) \in <_{\widehat{P}_M} \tag{C9}$$

and with Lemma 4 and the definition of $\prec$

$$\forall \, c, d : Op \times Val, t^+ : ext(t) \bullet (res(c), inv(d)) \in <_{comp(t^+)} \Rightarrow (res(c), inv(d)) \in <_{\widehat{P}_M} \tag{C10}$$

Since $<_{P_M}$ allows t'' (C7) and with the definition of $\Subset$

$$\forall \, c, d : Op \times Val \bullet (res(c), inv(d)) \in <_{\widehat{P}_M} \wedge inv(d) \in events(t'') \Rightarrow (res(c), inv(d)) \in <_{t''} \tag{C11}$$

From (C5) we know that $\forall \, d : Op \times Val \bullet inv(d) \in events(comp(t^+)) \Rightarrow inv(d) \in events(t'')$ and hence from (C10) and (C11) we have

$$\forall \, c, d : Op \times Val \bullet (res(c), inv(d)) \in <_{comp(t^+)} \Rightarrow (res(c), inv(d)) \in <_{t''} \tag{C12}$$

which is $\prec_{comp(t^+)} \subseteq \prec_{t''}$ and hence we also have

$$\prec_{comp(t^+)_{|ir}} \subseteq \prec_{t''_{|ir}} \tag{C13}$$

Since $t^+ \in ext(t)$, and $h = t_{|ir}$, there exists a $h^+ \in ext(h)$ such that

$$h^+ = t^+_{|ir} \tag{C14}$$

From (C14) and (C13), it follows that

$$\prec_{comp(h^+)} \subseteq \prec_{t''_{|ir}} \tag{C15}$$

and from (C8)

$$\prec_{comp(h^+)} \subseteq \prec_{h'} \tag{C16}$$

This is the second conjunct of the consequent. We now derive the first.

From (C8) and (C5) we have

$$comp(t^+)_{|ir} \sim h' \tag{C17}$$

and hence with (C14)

$$comp(h^+) \sim h' \tag{C18}$$

Thus conjoining (C18) and (C16) gives us

$$comp(h^+) \sim h' \ \wedge \ \prec_{comp(h^+)} \subseteq \prec_{h'} \tag{C19}$$

which is the consequent of the theorem. $\qquad\square$

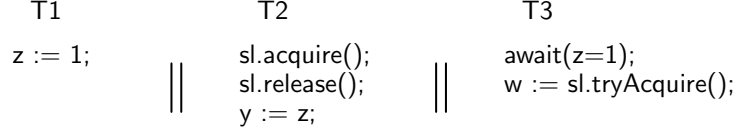| T1		T2		T3

z := 1;	‖	sl.acquire();
		sl.release();
		y := z;
			‖	await(z=1);
				w := sl.tryAcquire();

Fig. 9. Program using spinlock

7. Example applications

To demonstrate the application of standard linearizability for proving (or disproving) correctness of concurrent objects, we discuss two typical examples, a spinlock [HS08] and a work-stealing deque [LPCZN13] (the latter developed specifically for ARM). Our soundness result means that whenever an implementation is incorrect with respect to a specification on a given weak memory model, we can use linearizability to prove this. Our completeness result means that whenever an implementation is correct with respect to a specification on a given weak memory model, again we can use linearizability to prove it.

7.1. Correctness on TSO

Consider a spinlock object with operations acquire, release and tryAcquire specified as follows.

acquire	release	tryAcquire
await(x := 1); x := 0	x := 1;	if (x=1) x := 0; return 1 else return 0

A typical concurrent implementation [HS08] which is correct on SC is

acquire	release	tryAcquire

while (true) {
  if (TAS(x, 1, 0)=1) return
  while (x=0) {}
}
	x := 1;	return TAS(x, 1, 0)

where $TAS(x,a,b)$ is the atomic hardware primitive test-and-set which, when x is a, sets x to b and returns 1, and otherwise returns 0. The TAS instruction has a built-in fence to ensure any change it makes to x is immediately visible to all threads.

An earlier version of linearizability on TSO [DSD14] proved that this implementation is correct on TSO. However (as discussed in Section 1) it did not require that abstract operations take effect immediately. Under our definition of object refinement (for which we assume abstract operations take effect atomically), it is not correct.

Consider the client program in Figure 9 which contains a triangular race. The program uses a spinlock object sl for which we assume that initially $x = 1$ and $z = 0$. Following the operational semantics of TSO in [SSO$^+$10], one possible trace of this program is[9]

$\langle inv(\mathsf{T2}, \mathsf{sl.acquire}), \bot), res((\mathsf{T2}, \mathsf{sl.acquire}, \bot), \mathit{eff}((\mathsf{T2}, \mathsf{sl.acquire}), \bot), inv((\mathsf{T2}, \mathsf{sl.release}), \bot),$
$res(\mathsf{T2}, \mathsf{sl.release}, \bot), step(\mathsf{T2}, \mathsf{y} := 0), step(\mathsf{T1}, \mathsf{z} := 1), \mathit{eff}(\mathsf{T1}, \mathsf{z} := 1), step(\mathsf{T3}, \mathsf{await}(z = 1)),$
$\mathit{eff}(\mathsf{T3}, \mathsf{await}(z = 1)), inv((\mathsf{T3}, \mathsf{sl.tryAcquire}), \bot), res((\mathsf{T3}, \mathsf{sl.tryAcquire}), 0), \mathit{eff}((\mathsf{T3}, \mathsf{sl.tryAcquire}), 0),$
$step(\mathsf{T3}, \mathsf{w} := 0), \mathit{eff}(\mathsf{T3}, \mathsf{w} := 0), \mathit{eff}((\mathsf{T2}, \mathsf{sl.release}), \bot), \mathit{eff}(\mathsf{T2}, \mathsf{y} := 0)\rangle$

This trace corresponds to thread T2 acquiring and releasing the lock and reading the initial value of z, but not flushing the value written to x by the release operation until after the other two threads have run to completion. The observable behaviour of the trace is

$\langle \mathit{eff}(\mathsf{T1}, \mathsf{z} := 1), \mathit{eff}(\mathsf{T3}, \mathsf{w} := 0), \mathit{eff}(\mathsf{T2}, \mathsf{y} := 0)\rangle$

---

[9] The effects in TSO correspond to flushes for program steps and operations which write to global variables, and immediately follow the program step or operation response otherwise.

| | T1 | | T2 |
| | sl.acquire();
y := y + 1;
sl.release() | $\parallel$ | sl.acquire();
y := y + 1;
sl.release() |

Fig. 10. Another program using spinlock

This is not an observable trace of the program running with an abstract object whose effects are not delayed: if y=0 then the step y := z and hence sl.release on T2 must have occurred before z := 1 on T1, and hence before sl.tryAcquire on T3. Hence, we do not have object refinement. We can prove this with the standard definition of linearizability due to it being sound with respect to object refinement.

The spinlock implementation without the tryAcquire operation is, however, known to be correct on TSO [SSO+10]. Again we can show this using our definition of object refinement.

The traces of the implementation derived from the operational semantics of TSO show that if an acquire has responded (and hence has been observed due to the fence in the TAS) then another acquire cannot respond until after a release on the same core has responded or a release on another core has been observed. This coincides with what can be observed from the abstract specification, i.e., an acquire is always followed by a release (before another acquire can occur). Hence, object refinement holds. Again we can prove this with the standard definition of linearizability due to it being complete with respect to object refinement.

7.2. Correctness on ARM and POWER

It is easy to show that the spinlock implementation of Section 7.1, even without the tryAcquire operation, is not correct on ARM and POWER using our definition of object refinement. For example, consider the client program in Figure 10 for which we assume that initially x = 1 and y = 0. Following the operational semantics of ARM and POWER given in [CS18], one possible trace of this program is[10]

$\langle inv((\mathsf{T1}, \mathsf{sl.acquire}), \bot), res((\mathsf{T1}, \mathsf{sl.acquire}), \bot), e\!f\!f((\mathsf{T1}, \mathsf{sl.acquire}), \bot), step(\mathsf{T1}, \mathsf{y} := 1),$
$inv((\mathsf{T1}, \mathsf{sl.release}), \bot), res((\mathsf{T1}, \mathsf{sl.release}), \bot), e\!f\!f((\mathsf{T1}, \mathsf{sl.release}), \bot), inv((\mathsf{T2}, \mathsf{sl.acquire}), \bot),$
$res((\mathsf{T2}, \mathsf{sl.acquire}), \bot), e\!f\!f((\mathsf{T2}, \mathsf{sl.acquire}), \bot), step(\mathsf{T2}, \mathsf{y} := 1), inv((\mathsf{T2}, \mathsf{sl.release}), \bot), res((\mathsf{T2}, \mathsf{sl.release}), \bot),$
$e\!f\!f((\mathsf{T2}, \mathsf{sl.release}), \bot), e\!f\!f(\mathsf{T1}, \mathsf{y} := 1), e\!f\!f(\mathsf{T2}, \mathsf{y} := 1)\rangle$

This trace corresponds to the response of T1's release operation being observed before its update to y. This allows T2's acquire to occur followed by its update of y before T1's new value of y is observable by T2. Hence, both threads update y to 1.

Since the observable behaviour $\langle e\!f\!f(\mathsf{T1}, \mathsf{y} := 1), e\!f\!f(\mathsf{T2}, \mathsf{y} := 1)\rangle$ of the above trace is not possible using the specification, the implementation is not correct on ARM or POWER: object refinement does not hold. We can prove this with the standard definition of linearizability due to it being sound with respect to object refinement.

7.2.1. Chase-Lev deque

As a more substantial example, consider the below code for a version of the Chase-Lev work-stealing deque (double-ended queue) [CL05] developed specifically for ARM [LPCZN13]. The code shown corresponds to a refactoring used in [CS18] which, for example, eliminates returns from within a branch, and for simplicity assumes the elements of the deque are integers.

---

[10] In the operational semantics of [CS18] the placement of effects can be derived from the model of the "storage subsystem" which keeps track of which updates to global variables have been seen by which threads.

```
put(v)                         take                               steal
   int t;                         int h,t,task;                      int h,t,task;
   t := tail;                     t := tail-1;                       h := head;
   tasks[t mod L] := v;           tail := t;                         fence;
   fence;                         fence;                             t := tail;
   tail := t+1                    h := head;                         cfence;
                                  if (h <= t)                        if (h < t)
                                     task := tasks[t mod L];            task := tasks[h mod L];
                                     if (h=t)                           cfence;
                                        if !CAS(head, h, h + 1) then     if !CAS(head, h, h+1)
                                           task := empty;                   task := fail;
                                        tail := tail+1;               else
                                  else                                   task := empty;
                                     task := empty;                   return task
                                     tail := tail+1;
                                  return task
```

The deque is implemented as a circular array of size L with a head and tail pointer. Elements may be *put* on or *taken* from the tail by a worker thread, and additionally, other (thief) processes may *steal* an element from the head of the deque (in order to balance system workload). Since the put and take operations are executed by a single thread, there is no interference between these two operations.

The put operation straightforwardly adds an element to the end of the deque, incrementing the tail pointer. It includes a full fence so that the increment of the tail pointer does not take effect before the element is placed in the array.

The interesting behaviour is in the way that the take and steal operations interact when called concurrently. To take the task at position t=tail-1, the worker process decrements tail to equal t, thereby publishing its intent to take that task. This publication, ensured by the following fence, means subsequent thief processes will not try to steal the task at position t. It then reads head into h and if h < t knows that there is more than one task in the deque and it is safe to take the task at position t, i.e., no thief process can concurrently steal it.

If t < h the worker knows the deque is empty and sets tail back to its original value. The final possibility is that h=t. In this case, there is one task on the deque and conflict with a thief may arise. To deal with this conflict, both the take and steal operations employ an atomic CAS (compare-and-swap) operation. An operation CAS(x,y,z) checks whether x equals y and, if so, updates x to z and returns true, otherwise it returns false leaving x unchanged. The CAS is atomic, and the update is immediately written to memory since the CAS operation also implements a fence.

If h=t, rather than decrementing tail to take the task, the worker increments head. Therefore, if the worker finds h=t, it restores tail to its original value. The steal operation works similarly. The operation reads the deque's head and tail into h and t, and if the deque is not empty tries to increment head from h to h+1 using a CAS. If it succeeds, the value of head has not been changed since read into the local variable h and hence the thief has stolen the task.

Additionally, the steal operation contains two control fence barriers (ctrl_isync in ARM), denoted cfence in code above. A control fence ensures all branch instructions occurring before it take effect before any *loads*, i.e., reads of global variables, occurring after it. It can therefore be used to avoid *speculative execution* of loads occurring in a branch. As shown in [CS18], the first control fence is redundant, and the second is incorrectly placed. Eliminating the first cfence and swapping the order of the second cfence with the preceding assignment gives the intended behaviour.

To show that we could also detect these problems using standard linearizability, we define the specification of the expected deque behaviour in terms of a sequence q and sequence operations $⌢$ (sequence concatenation), head, tail, last and front. We also use $\sqcap$ to denote nondeterministic choice, which is required to model that steal may fail.

$$\begin{array}{cc} \text{T1} & \text{T2} \\ \text{d.put(1);} \parallel & \text{y := d.steal()} \end{array}$$

Fig. 11. Program using the Chase-Lev deque

$$\begin{array}{ccc} \text{T1} & \text{T2} & \text{T3} \\ \text{z := 1;} \parallel & \text{d.put(1);} & \parallel \text{await(z=1);} \\ & \text{y := z;} & \text{w := d.steal();} \end{array}$$

Fig. 12. Another program using the Chase-Lev deque

```
put(v)                take                        steal
   q := q⌢⟨ v⟩           if q=⟨ ⟩                    if q=⟨ ⟩
                            return empty                return empty
                         else                        else
                            q := front(q);              q := tail(q); return head(q)
                            return last(q)           ⊓
                                                     return fail
```

Consider the simple program in Figure 11 in which the deque d is initially empty and the circular buffer implementing it initially holds only zeroes. An example trace of the program is

$\langle inv((\mathsf{T2}, \mathsf{d.steal}), \bot), inv((\mathsf{T1}, \mathsf{d.put}), 1), res((\mathsf{T1}, \mathsf{d.put}), \bot), \mathit{eff}((\mathsf{T1}, \mathsf{d.put}), \bot), \mathit{eff}((\mathsf{T2}, \mathsf{d.steal}), 0),$
$res((\mathsf{T2}, \mathsf{d.steal}), 0), step(\mathsf{T2}, \mathsf{y} := 0), \mathit{eff}(\mathsf{T2}, \mathsf{y} := 0)\rangle$

This trace can occur since the load, $\mathsf{t} := \mathsf{tail}$, in steal does not need to take effect before the first control fence, and the load after the if instruction can take effect before the if statement, so that the code between the full fence and the second control fence is effectively

cfence; t := tail; task := tasks[h mod L]; if(h < t) .

Since the two loads are also not required to take effect in order on ARM, this can be effectively executed as

cfence; task := tasks[h mod L]; t := tail; if(h < t) .

Now assume that the steal operation is invoked and reaches the load, $\mathsf{task} := \mathsf{tasks}[\mathsf{h}\ \mathsf{mod}\ \mathsf{L}]$. Since nothing has been put into the deque, task will be set to the initial value at h which is 0. If the put operation is then invoked and takes effect, when the steal continues it will load the new value of tail into t. Since this new value is greater than h, and no other operation will change h in the program of Figure 11, the steal operation will complete returning 0.

The observable behaviour of this trace

$\langle \mathit{eff}(\mathsf{T2}, \mathsf{y} := 0)\rangle$

is not a possible observable trace of the program running with an abstract object: the only observable traces are $\langle \mathit{eff}(\mathsf{T2}, \mathsf{y} := \mathsf{empty})\rangle$ and $\langle \mathit{eff}(\mathsf{T2}, \mathsf{y} := 1)\rangle$. Hence, the failure of the implementation to meet the specification could have been detected using standard linearizability.

As suggested in [CS18], the implementation can be fixed by moving the second control fence before the load into task. This ensures the load into task takes effect after the if statement, and hence also after the load into t (the latter cannot take effect after the if statement according to the semantics in [CS18]). Hence, task in steal can only be set to empty or 1, and the only observable behaviours of the program are those satisfying the specification.

Consider, however, the program containing a triangular race in Figure 12. On ARM the await instruction includes a fence to prevent subsequent loads taking effect earlier than it. A possible behaviour of this program when executed with the fixed implementation is

$\langle inv(\mathsf{T2}, \mathsf{d.put}), \bot), res(\mathsf{T2}, \mathsf{d.put}, \bot), step(\mathsf{T2}, \mathsf{y} := 0), step(\mathsf{T1}, \mathsf{z} := 1), \mathit{eff}(\mathsf{T1}, \mathsf{z} := 1), step(\mathsf{T3}, \mathsf{await}(\mathsf{z} = 1)),$
$\mathit{eff}(\mathsf{await}(\mathsf{z} = 1)), inv((\mathsf{T3}, \mathsf{d.steal}), \bot), res((\mathsf{T3}, \mathsf{d.steal}), \mathsf{empty}), \mathit{eff}((\mathsf{T3}, \mathsf{d.steal}), \mathsf{empty}),$
$step(\mathsf{T3}, \mathsf{w} := \mathsf{empty}), \mathit{eff}(\mathsf{T3}, \mathsf{w} := \mathsf{empty}), \mathit{eff}((\mathsf{T2}, \mathsf{d.put}), \bot), \mathit{eff}(\mathsf{T2}, \mathsf{y} := 0)\rangle$

This occurs when the effect of the put operation is delayed until after the steal operation has occurred. Specifically, tail is not updated until after the steal operation. The observable behaviour is

$$\langle \mathit{eff}\,(\mathsf{T1}, \mathsf{z} := 1),\, \mathit{eff}\,(\mathsf{T3}, \mathsf{w} := \mathsf{empty}),\, \mathit{eff}\,(\mathsf{T2}, \mathsf{y} := 0)\rangle$$

which is not a possible behaviour according to the specification. Hence, this would be detected when attempting to prove standard linearizability. It could be fixed by adding a fence at the end of the put operation. Without this additional fence, the program of Figure 12 can produce the unexpected observable behaviour above. If this behaviour is deemed acceptable (as it is harmless if a thief process returns empty when there is in fact an element in the deque) then to prove correctness would require a modification of the specification to allow steal to nondeterministically return empty in such circumstances.

The need for a fence at the end of the put operation highlights a consequence of our notion of correctness. As shown in Figure 12, any operation whose effect can influence the outcome of another operation needs to be fenced to be correct on a weak memory model. Note however that this does not mean all operations require fences. One example of an operation where a fence is not required is the release operation of spinlock on TSO. Also, both the take and steal operations of the Chase-Lev deque on ARM do not need to be fenced when they fail to return a task. In particular, it is not a problem when take does not immediately restore tail to its original value before the next steal. Restoring tail's original value is only required in take when the deque is empty, and if another put occurs then tail will be correctly updated by the fence in put (since put and take are both only called by the same worker thread).

8. Related work

Traditionally, trace refinement provides the notion of correctness for programs: an implementation is correct with respect to its specification if and only if each observable behaviour of the implementation can also be observed from the specification [BvW94, Bac90, AL91]. In recent work [SW17] we showed that trace refinement and linearizability do not coincide; linearizability is, in general, weaker than trace refinement. This is because in the context of concurrent objects that are called by a client program, an object implementation is deemed correct if and only if the client program cannot differentiate between the object implementation and its abstract specification. This notion of correctness differs from trace refinement.

To capture what is observed by client programs under *sequentially consistent* (SC) architectures (i.e., those without a weak memory model), the notions of *observational refinement* [FORY10] and *contextual refinement* [DG16] have been introduced. However, both of these definitions do not provide a notion of correctness under weak memory models, as they do not take into account that an event occurring on one thread might be observable later on another. Instead events become observable immediately after their occurrence. Our definition of object refinement provides a correctness notion for concurrent objects which is applicable to weak memory models.

Bouajjani et al. [BMM11, BDM13] introduce a notion of *robustness* which requires each trace of a concurrent implementation running on a weak memory model (specifically, TSO) to be equivalent to a trace of the implementation running on a sequentially consistent (SC) architecture. This is a stronger requirement than object refinement which requires that each trace of a concurrent implementation has the same *observable behaviour* as a trace of a specification. This allows the implementation to exhibit non-SC behaviour provided that it cannot be observed by any client program. The release operation of spinlock on TSO, and the take and steal operations of the Chase-Lev deque on ARM provide examples of where non-SC behaviour is allowed. Furthermore, robustness checks a correspondence between a given implementation on two different memory models (a weak memory model and SC). Object refinement, on the other hand, checks a correspondence between an implementation and a specification. If the specification is sufficiently weak then the implementation may exploit non-SC behaviour to implement it.

Two recent publication are related to our result: Dongol et al. [DJRA18] investigate the effects of hardware memory models like TSO, ARM and POWER, and Doherty et al. [DDWD18] investigate linearizability for concurrent objects under C11.

Dongol et al. [DJRA18] define *real-time hb-linearisability* as the correctness condition for concurrent objects running on multicore processors. This differs from our result due to the way the semantics of object

implementations and specifications are characterised. Similar to our work, traces follow a real-time ordering of events which (on one thread) corresponds to the program order. Additionally, however, the events in each trace (including the output values) have to satisfy the happens-before ordering hb (as defined in [AMT14]) postulated for the weak memory model. For the abstract specification an additional *specification order* is introduced that ensures the intended sequences of events (and output values). *Real-time hb-linearisability* extends standard linearizability with an additional check that the specification order is maintained between matching abstract and concrete histories.

In our work, both the happens-before order and the specification order are incorporated into the semantics of concrete and abstract objects. Instead of introducing an additional check in the linearizability definition (and hence strengthening it), we abstract from the ordering constraints of M and assume that $[\![A]\!]$ contains only those histories that satisfy the specification order (and produce the intended output values) and similarly $[\![C]\!]_M$ contains only histories in which sequences of events and their output values satisfy the constraints imposed by the object's code and the memory model.

The benefits of this viewpoint are threefold: Firstly we can observe that standard linearizability is sound and complete with respect to object refinement (no strengthening is required), and standard linearizability is known to be compositional [HW90]. Secondly, our semantic model is generic and hence independent of any particular weak memory model semantics (whereas [DJRA18] is linked to the semantic model of [AMT14]). Thirdly, the assumptions we make on program/object semantics under weak memory models are formalised as simple axioms (see Sections 3 and 4). These axioms are generic in that they are based on principles of object encapsulation and abstract specification – no reasoning about weak memory behaviour is required at this level.

Dongol et al. also define *causal hb-linearisability* for processes running on distributed systems (and potentially future multicore architectures). This definition, motivated by requiring programs to have less synchronisation overhead, removes the need to preserve the real-time order on events between the abstract and concrete specifications, preserving instead the happens-before order. As outlined in [DJRA18] this more relaxed definition allows us to prove objects such as that in our Figure 3 (repeated below) linearizable.

$$\begin{array}{c|c} \text{o.WriteX;} & \text{o.WriteY;} \\ \text{z := o.ReadY} & \text{w := o.ReadX} \end{array}$$

As discussed in Section 1 this program can produce the result z=w=0 on TSO when the WriteX and WriteY operations write value 1 to variable x and value 1 to variable y, respectively. This does not match any behaviour of the abstract object whose operations are viewed as taking effect immediately. Hence causal hb-linearisability is not sound with respect to our definition of object refinement.

Doherty et al. [DDWD18] define *causal linearizability* as their correctness condition which is closely related to causal hb-linearisability [DJRA18] but improves on its compositionality result. Whilst causal hb-linearisability is only compositional under certain constraints either on the abstract specification or on the client program, causal linearizability is proved to be compositional for any specification and under any client. As with causal hb-linearisability, however, it does not target hardware weak memory models, but the C11 memory model. Consequently, it also is not sound with respect to our definition of object refinement.

References

[AFI+08] J. Alglave, A. Fox, S. Ishtiaq, M. O. Myreen, S. Sarkar, P. Sewell, and F.Z. Nardelli. The Semantics of Power and ARM Multiprocessor Machine Code. In L. Petersen and M.M.T. Chakravarty, editors, *DAMP '09*, pages 13–24. ACM, 2008.

[AL91]   M. Abadi and L. Lamport. The existence of refinement mappings. *Theoretical Computer Science*, 82(2):253–284, 1991.

[AMT14]   J. Alglave, L. Maranget, and M. Tautschnig. Herding cats: Modelling, simulation, testing, and data mining for weak memory. *ACM Trans. Program. Lang. Syst.*, 36(2):7:1–7:74, 2014.

[Bac90]   R.-J.R. Back. Refinement calculus, part II: Parallel and reactive programs. In *Stepwise Refinement of Distributed Systems Models, Formalisms, Correctness*, pages 67–93. Springer, 1990.

[BDM13]   A. Bouajjani, E. Derevenetc, and R. Meyer. Checking and enforcing robustness against TSO. In M. Felleisen and P. Gardner, editors, *Programming Languages and Systems (ESOP 2013)*, pages 533–553. Springer, 2013.

[BGMY12]   S. Burckhardt, A. Gotsman, M. Musuvathi, and H. Yang. Concurrent library correctness on the TSO memory model. In H. Seidl, editor, *ESOP 2012*, volume 7211 of *LNCS*, pages 87–107. Springer, 2012.

[BMM11]   A. Bouajjani, R. Meyer, and E. Möhlmann. Deciding robustness against total store ordering. In Luca Aceto, Monika Henzinger, and Jiří Sgall, editors, *Automata, Languages and Programming*, pages 428–440. Springer, 2011.

[BOS⁺11]   M. Batty, S. Owens, S. Sarkar, P. Sewell, and T. Weber. Mathematizing C++ concurrency. In *POPL*, pages 55–66. ACM, 2011.

[BvW94]   R.-J.R. Back and J. von Wright. Trace refinement of action systems. In *CONCUR '94*, volume 836 of *LNCS*, pages 367–384. Springer, 1994.

[CL05]   D. Chase and Y. Lev. Dynamic circular work-stealing deque. In *SPAA'05: Proceedings of the 17th annual ACM symposium on Parallelism in algorithms and architectures*, pages 21–28, New York, NY, USA, 2005. ACM Press.

[CS18]   R.J. Colvin and G. Smith. A wide-spectrum language for verification of programs on weak memory models. In K. Havelund, J. Peleska, B. Roscoe, and E. de Vink, editors, *FM 2018*, volume 10951 of *LNCS*, pages 240–257. Springer, 2018.

[DD16]   S. Doherty and J. Derrick. Linearizability and causality. In *SEFM 2016*, volume 9763 of *LNCS*, pages 45–60. Springer, 2016.

[DDGS15]   B. Dongol, J. Derrick, L. Groves, and G. Smith. Defining correctness conditions for concurrent objects in multicore architectures. In *ECOOP '15*, LIPIcs, pages 470–494. Schloss Dagstuhl – Leibnis-Zentrum für Informatik, 2015.

[DDWD18]   S. Doherty, B. Dongol, H. Wehrheim, and J. Derrick. Making linearizability compositional for partially ordered executions. In C. A. Furia and K. Winter, editors, *Integrated Formal Methods (iFM 2018)*, volume 11023 of *Lecture Notes in Computer Science*, pages 110–129. Springer, Cham, 2018.

[DG16]   B. Dongol and L. Groves. Contextual trace refinement for concurrent objects: Safety and progress. In K. Ogata, M. Lawford, and S. Liu, editors, *ICFEM 2016*, pages 261–278. Springer, 2016.

[DJRA18]   B. Dongol, R. Jagadeesan, J. Riely, and A. Armstrong. On abstraction and compositionality for weak-memory linearisability. In I. Dillig and J. Palsberg, editors, *VMCAI'18*, volume 10747 of *LNCS*, pages 183–204. Springer, 2018.

[DS15]   J. Derrick and G. Smith. A framework for correctness criteria on weak memory models. In N. Bjørner and F.S. de Boer, editors, *FM 2015*, volume 9109 of *LNCS*, pages 178–194. Springer, 2015.

[DSD14]   J. Derrick, G. Smith, and B. Dongol. Verifying linearizability on TSO architectures. In E. Albert and E. Sekerinski, editors, *iFM 2014*, volume 8739 of *LNCS*, pages 341–356. Springer, 2014.

[DSGD14]   J. Derrick, G. Smith, L. Groves, and B. Dongol. Using coarse-grained abstractions to verify linearizability on TSO architectures. In E. Yahav, editor, *HVC 2014*, pages 1–16. Springer, 2014.

[DSGD17]   J. Derrick, G. Smith, L. Groves, and B. Dongol. A proof method for linearizability on TSO architectures. In M. Hinchey, J.P. Bowen, and E.-R. Olderog, editors, *Provably Correct Systems*, pages 61–91. Springer, 2017.

[DSW11]   J. Derrick, G. Schellhorn, and H. Wehrheim. Mechanically verified proof obligations for linearizability. *ACM Trans. Program. Lang. Syst.*, 33(1):4:1–4:43, 2011.

[FGP⁺16]   S. Flur, K.E. Gray, C. Pulte, S. Sarkar, A. Sezgin, L. Maranget, W. Deacon, and P. Sewell. Modelling the ARMv8 architecture, operationally: Concurrency and ISA. In R. Bodik and R. Majumdar, editors, *POPL 2016*, pages 608–621. ACM, 2016.

[FORY10]   I. Filipović, P. W. O'Hearn, N. Rinetzky, and H. Yang. Abstraction for concurrent objects. *Theoretical Computer Science*, 411(51-52):4379 – 4398, 2010.

[GMY12]   A. Gotsman, M. Musuvathi, and H. Yang. Show no weakness: Sequentially consistent specifications of TSO libraries. In M. Aguilera, editor, *DISC 2012*, volume 7611 of *LNCS*, pages 31–45. Springer, 2012.

[GY11]   A. Gotsman and H. Yang. Liveness-preserving atomicity abstraction. In *ICALP 2011*, volume 6756 of *LNCS*, pages 453–465. Springer, 2011.

[HS08]   M. Herlihy and N. Shavit. *The Art of Multiprocessor Programming*. Morgan Kaufmann, 2008.

[HW90]   M. Herlihy and J. M. Wing. Linearizability: A correctness condition for concurrent objects. *ACM Trans. Program. Lang. Syst.*, 12(3):463–492, 1990.

[LPCZN13]   N.M. Lê, A. Pop, A. Cohen, and F. Zappa Nardelli. Correct and efficient work-stealing for weak memory models. In *Proceedings of the 18th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '13, pages 69–80, New York, NY, USA, 2013. ACM.

[MHMS⁺12]   S. Mador-Haim, L. Maranget, S. Sarkar, K. Memarian, J. Alglave, S. Owens, R. Alur, M.M.K. Martin, P. Sewell, and D. Williams. An axiomatic memory model for POWER multiprocessors. In *CAV'12*, pages 495–512, Berlin, Heidelberg, 2012. Springer-Verlag.

[MS04]   M. Moir and N. Shavit. Concurrent data structures. *Handbook of Data Structures and Applications*, pages 47:1–47:30, 2004.

[NMS16]   K. Nienhuis, K. Memarian, and P. Sewell. An operational semantics for C/C++11 concurrency. In *OOPSLA*, pages 111–128. ACM, 2016.

[Owe10]   S. Owens. Reasoning about the implementation of concurrency abstractions on x86-TSO. In T. D'Hondt, editor, *ECOOP 2010*, volume 6183 of *LNCS*, pages 478–503. Springer, 2010.

[SSA⁺11]   S. Sarkar, P. Sewell, J. Alglave, L. Maranget, and D. Williams. Understanding POWER multiprocessors. *SIGPLAN Not.*, 46(6):175–186, June 2011.

[SSO⁺10]   P. Sewell, S. Sarkar, S. Owens, F.Z. Nardelli, and M.O. Myreen. x86-TSO: a rigorous and usable programmer's model for x86 multiprocessors. *Commun. ACM*, 53(7):89–97, 2010.

[SW17]   G. Smith and K. Winter. Relating trace refinement and linearizability. *Formal Aspects of Computing*, 29(6):935–950, 2017.

[SWC18]   G. Smith, K. Winter, and R.J. Colvin. Correctness of concurrent objects under weak memory models. In J. Derrick, B. Dongol, and S. Reeves, editors, *Refine 2018*, volume 282 of *EPTCS*, pages 53–67. Open Publishing Association, 2018.

[TMW13]   O. Travkin, A. Mütze, and H. Wehrheim. SPIN as a linearizability checker under weak memory models. In V. Bertacco and A. Legay, editors, *HVC2013*, volume 8244 of *LNCS*, pages 311–326. Springer, 2013.

[TW14]   O. Travkin and H. Wehrheim. Handling TSO in mechanized linearizability proofs. In E. Yahav, editor, *HVC2014*, volume 8855 of *LNCS*, pages 132–147. Springer, 2014.

A. Proof of lemmas

A.1. Proof of Lemma 2

Lemma 2. If the events of a trace t are events of a program P then so are the events of any completion of t.

$$\forall P \bullet \forall t : Trace \bullet \forall t^+ : ext(t) \bullet events(t) \subseteq events(P) \Rightarrow events(comp(t^+)) \subseteq events(P)$$

Proof
Let P be a program, $t \in Trace$ and $t^+ \in ext(t)$.
With the definition of $comp$ we have that $events(comp(t^+)) \subseteq events(t^+)$.
With the definition of ext it follows that $\exists tr : seq\ Event \bullet$

(a)　　　$t \frown tr \in Trace\ \wedge$
(b)　　　$(\forall i < \#tr \bullet \exists c : Op \times Val \bullet tr_i = res(c))\ \wedge$
(c)　　　$t^+ = t \frown tr$

From (c) we can deduce that $events(t^+) = events(t) \cup events(tr)$,
and from (b) that $\forall b : events(tr) \bullet \exists (op, out) : Op \times Val \bullet b = res(op, out)$.
With the definition of $Trace$ and (a) it follows that $\exists in : Value \bullet inv(op, in) \in events(t)$.
From this and Axiom (1) we can derive that $events(t) \subseteq events(P) \Rightarrow events(tr) \subseteq events(P)$,
and with $events(t^+) = events(t) \cup events(tr)$ we have that
$events(t) \subseteq events(P) \Rightarrow events(t^+) \subseteq events(P)$.
From this and $events(comp(t^+)) \subseteq events(t^+)$ we can conclude that
$events(t) \subseteq events(P) \Rightarrow events(comp(t^+)) \subseteq events(P)$.　　　□

A.2. Proof of Lemma 3

Lemma 3 states that if a trace is allowed by a program then so is any completion that only adds responses for operations whose effects occur in the trace. The proof of this lemma relies on two sub-results which refer to the construction of a trace's completion, the addition of responses and the elimination of pending invocations.

In the following proofs, $S \setminus T$ denotes the set S minus the elements of set T.

Lemma 3a. If a trace t is allowed by P on M, then so is any trace formed by adding a response to t when the corresponding effect occurs in t.

$$\forall P, M \bullet \forall res(c) : events(P);\ t : Trace \bullet (\exists i \le \#t \bullet t_i = \mathit{eff}(c)) \wedge <_{P_M} \Subset <_t \Rightarrow <_{P_M} \Subset <_{t \frown \langle res(c) \rangle}$$

Proof
From the definition of $<_{t \frown \langle res(c) \rangle}$ it follows that $\forall (a, b) : <_{t \frown \langle res(c) \rangle} \setminus <_t \bullet b = res(c)$.
Let $c = (a, out)$ where $inv(a, in)$ and $res(a, out)$ are in $events(P)$, and let e be any event in $events(P)$ such that $(e, res(a, out)) \in <_{P_M}$. For $<_{P_M} \Subset <_{t \frown \langle res(a, out) \rangle}$ to be true, we need to show that e is in t.

Case 1: $e \ne \mathit{eff}(a, out)$
　　　With Axiom (4) it follows that $(e, inv(a, in)) \in <_{P_M}$.
　　　Since $t \frown \langle res(a, out) \rangle \in Trace$ we know that $inv(a, in)$ is in t and hence if $<_{P_M} \Subset <_t$ (i.e.,
　　　the second conjunct of the antecedent holds), we can conclude that e is in t.

Case 2: $e = \mathit{eff}(a, out)$
　　　If $\exists i \le \#t \bullet t_i = \mathit{eff}(c)$ (i.e., the first conjunct of the antecedent holds), then e is in t.　　　□

Following Lemma 3a., if a trace t is allowed by a program P on memory model M, the trace formed by extending t with a sequence of responses whose corresponding effects are in t is also allowed by P on M.

Corollary 3.1

$$\forall P, M \bullet \forall t : Trace \bullet \forall t \frown tr : ext(t) \bullet$$
$$(\forall i \leq \#tr;\ c : Op \times Val \bullet tr_i = res(c) \Rightarrow \exists j \leq \#t \bullet t_j = e\!f\!f(c)) \wedge <_{P_M} \Subset <_t \Rightarrow <_{P_M} \Subset <_{t \frown tr}$$

Lemma 3b. If, after removing an invocation from a trace, the resulting event sequence is still a trace, i.e., the invocation was a pending invocation with neither a response or effect in the trace, then if the original trace is allowed by a program P on memory model M, the resulting trace is allowed by P on M.

$$\forall P, M \bullet \forall inv(c) : events(P) \bullet \forall t \frown \langle inv(c) \rangle \frown t' : Trace \bullet$$
$$t \frown t' \in Trace \wedge <_{P_M} \Subset <_{t \frown \langle inv(c) \rangle \frown t'} \Rightarrow <_{P_M} \Subset <_{t \frown t'}$$

Proof
Let P be a program, M be a memory model, $t \frown \langle inv(c) \rangle \frown t' \in Trace$, $<_{P_M} \Subset <_{t \frown \langle inv(c) \rangle \frown t'}$ and $t \frown t' \in Trace$.

We can deduce that $\forall a, b : Events \bullet (a, b) : <_{t \frown \langle inv(c) \rangle \frown t'} \setminus <_{t \frown t'} \Rightarrow inv(c) \in \{a, b\}$.

Case 1: For all $e \in Event$ where $(e, inv(c)) \in <_{P_M}$, with the definition of $\Subset$, $(e, inv(c))$ is only enforced by P_M in traces that contain event $inv(c)$. Since $inv(c) \notin events(t \frown t')$ trace $t \frown t'$ is not affected by the enforced ordering of $(e, inv(c))$.

Case 2: For all $e \in Event$ and $inv(a, in), res(a, out) \in events(P)$ where $(inv(a, in), e) \in <_{P_M}$, if $e \neq e\!f\!f(a, out)$ then with Axiom (3) we also have $(res(a, out), e) \in <_{P_M}$.

Hence if $e \in events(t \frown \langle inv(a, in) \rangle \frown t')$ then, when $<_{P_M} \Subset <_{t \frown \langle inv(a, in) \rangle \frown t'}$, we have $e \in events(t')$ and also that $res(a, out) \in events(t')$. Hence $t \frown t' \notin Trace$ which contradicts the assumption.

If $e = e\!f\!f(a, out)$ and $e \in events(t \frown \langle inv(a, in) \rangle \frown t')$ then, by Axiom 2, it is in t'. Hence $t \frown t' \notin Trace$ which again contradicts the assumption.

On the other hand, if $e \notin events(t \frown \langle inv(a, in) \rangle \frown t')$ then $e \notin events(t \frown t')$, and the ordering of $(inv(a, in), e)$ is not enforced on trace $t \frown t'$.

From the above we can conclude that $<_{P_M} \Subset <_{t \frown t'}$. □

Following Lemma 3b., if a trace t is allowed by a program P on memory model M, the trace formed by removing all pending invocations from t is allowed by P on M.

Corollary 3.2

$$\forall P, M \bullet \forall t : Trace \bullet <_{P_M} \Subset <_t \Rightarrow <_{P_M} \Subset <_{comp(t)}$$

With the above two corollaries we can now prove Lemma 3.

Lemma 3. If a trace t is allowed by a program P on memory model M then so is any completion of t.

$$\forall P, M \bullet \forall t : Trace \bullet \forall t \frown tr : ext(t) \bullet$$
$$(\forall i \leq \#tr;\ c : Op \times Val \bullet tr_i = res(c) \Rightarrow \exists j \leq \#t \bullet t_j = e\!f\!f(c)) \wedge <_{P_M} \Subset <_t \Rightarrow <_{P_M} \Subset <_{comp(t \frown tr)}$$

Proof
For any P, M, $t \in Trace$ and $t \frown tr \in ext(t)$ where tr only includes responses for operations whose effects occur in t, with Corollary 3.2 we have $<_{P_M} \Subset <_{t \frown tr} \Rightarrow <_{P_M} \Subset <_{comp(t \frown tr)}$. From this and Corollary 3.1 we can deduce that $<_{P_M} \Subset <_t \Rightarrow <_{P_M} \Subset <_{comp(t \frown tr)}$. □

A.3. Proof of Lemma 4

Lemma 4 states that the operation order of the completion of a trace is a subset of the operation order of the original trace. The proof of Lemma 4 is broken into two steps according to the two steps of the construction of the completion of a trace. The first result states that additional responses that occur in a completion of a trace do not cause a modification of the order of operations. A second result states that the removal of pending invocation events does not affect the order of the remaining operations.

Lemma 4a. Adding a response event to a trace t does not affect the operation order $\prec_t$.

$$\forall\, c : Op \times Val \bullet \forall\, t ^\frown \langle res(c) \rangle : Trace \bullet \prec_t \,=\, \prec_{t ^\frown \langle res(c) \rangle}$$

Proof
Since $\prec_t$ only includes pairs $(res(c), inv(d))$, i.e., where $res(c)$ occurs in trace t before $inv(d)$, adding a response at the end of t does not change $\prec_t$. □

As a consequence of Lemma 4a we can conclude that extending a trace t with a sequence of response events does not affect $\prec_t$.

Corollary 4.1

$$\forall\, t : Trace \bullet \forall\, t^+ : ext(t) \bullet \prec_t \,=\, \prec_{t^+}$$

Lemma 4b. If, after removing an invocation from a trace, the resulting event sequence is still a trace, i.e., the invocation was a pending invocation, then its $\prec$ order is a subset of that of the original trace.

$$\forall\, t ^\frown \langle inv(c) \rangle ^\frown t' : Trace \bullet t ^\frown t' \in Trace \Rightarrow \prec_{t ^\frown t'} \subseteq \prec_{t ^\frown \langle inv(c) \rangle ^\frown t'}$$

Proof
For any $t ^\frown \langle inv(c) \rangle ^\frown t' \in Trace$, assume $t ^\frown t' \in Trace$.
From the definition of $Trace$ we have $inv(c) \notin events(t) \wedge inv(c) \notin events(t')$.
Then from the definition of $\prec_{t ^\frown t'}$ it follows that $\prec_{t ^\frown t'} \,=\, \prec_{t ^\frown \langle inv(c) \rangle ^\frown t'} \setminus \{ (a, b) :\, <_{t ^\frown \langle inv(c) \rangle ^\frown t'} \mid b = inv(c) \}$.
Hence, we can deduce that $\prec_{t ^\frown t'} \subseteq \prec_{t ^\frown \langle inv(c) \rangle ^\frown t'}$. □

Following Lemma 4b., removing all pending invocations from a trace results in a trace whose $\prec$ order is a subset of that of the original trace.

Corollary 4.2

$$\forall\, t : Trace \bullet \prec_{comp(t)} \subseteq \prec_t$$

The two corollaries provide us now with a succinct proof of Lemma 4.

Lemma 4. The $\prec$ order of a completion of a trace t is a subset of that of t.

$$\forall\, t : Trace \bullet \forall\, t^+ : ext(t) \bullet \prec_{comp(t^+)} \subseteq \prec_t$$

Proof
Let $t \in Trace$ and $t^+ \in ext(t)$. With Corollary 4.2 it follows that $\prec_{comp(t^+)} \subseteq \prec_{t^+}$ and together with Corollary 4.1 we can conclude that $\prec_{comp(t^+)} \subseteq \prec_t$. □

B. Transformation function proofs

According to case (T4) in Definition 4 the order of program events in t' is the same as in $comp(t^+)$.

$$\forall\, a, b : \mathcal{P}rogEvents \bullet (a, b) \in\, <_{comp(t^+)} \Rightarrow (a, b) \in\, <_{t'} \tag{i}$$

Operation events may be reordered in t' compared to $comp(t^+)$ unless their order in h' is the same as that in $comp(t^+)$:

- according to case (T1) an invocation will be placed at the same place in t' if $comp(t^+)$ and h' coincide on that event.
- according to case (T2), invocations in $comp(t^+)$ that do not match the head of h' will be placed into the remainder sequence r, whose content will be added later to t'.
- according to step (T5), an invocation which has been placed into the remainder sequence r can be placed in t' when it matches the event at the head of h'. Its occurrence in t' will be later than in $comp(t^+)$.
- according to case (T3) the occurrence of responses and effects in $comp(t^+)$ will be discarded and instead these events will be placed directly after their invocation event ((T1) and (T5)) to mimic atomicity of the abstract operation.
- note that by (T4) program events will only be placed in t' if invocations at the head of h' and r do not match, i.e., placing of object events has precedence over placing of program events.

It follows that, with respect to program events, invocations will either stay where they are or will be moved to a later place in t'. It is never the case that an invocation will be placed at an earlier place in t' (invocations only ever move to the right) w.r.t. program events.

$$\forall x : \mathcal{P}rogEvents, c : Op \times Val \bullet (x, inv(c)) \in <_{comp(t^+)} \Rightarrow (x, inv(c)) \in <_{t'} \qquad \text{(ii)}$$

With respect to program events, responses and operation effects will always be placed earlier in t' but never later than their placement in $comp(t^+)$. Responses and operation effects only ever move to the left w.r.t. program events. If the invocation stays where it is in the order, then its response and effect will move to an earlier point (due to (T1) and (T5)). If the invocation event is placed at a later point it will never be placed after a program event that followed its response event (due to the fact that the placement of object events has precedence over the placement of program events).

$$\forall x : \mathcal{P}rogEvents, c : Op \times Val \bullet (res(c), x) \in <_{comp(t^+)} \Rightarrow (res(c), x) \in <_{t'} \qquad \text{(iii)}$$

Due to (T1) and (T5) operation effects will be placed at the earliest point in t' (i.e., directly before the response event). Hence, with (iii) it follows that

$$\forall x : \mathcal{P}rogEvents, c : Op \times Val \bullet (eff(c), x) \in <_{comp(t^+)} \Rightarrow (eff(c), x) \in <_{t'} \qquad \text{(iv)}$$

The above properties of $trans$, (i)–(iv), and Axioms (3)–(6) of $<_{P_M}$ (on page 9) enable us to prove the four cases which show that P_M allows the constructed trace t', i.e., $<_{P_M} \Subset <_{t'}$, from which we can conclude that $t' \in \llbracket P \rrbracket_M$.

a.) $\quad \forall a, b : \mathcal{P}rogEvents \bullet (a, b) \in <_{P_M} \wedge b \in events(t') \Rightarrow (a, b) \in <_{t'}$

Proof
Since $<_{P_M} \Subset <_{comp(t^+)}$ (S4)
$\forall a, b : \mathcal{P}rogEvents \bullet (a, b) \in <_{P_M} \wedge b \in events(comp(t^+)) \Rightarrow (a, b) \in <_{comp(t^+)}$
with (i) we have
$\forall a, b : \mathcal{P}rogEvents \bullet (a, b) \in <_{P_M} \wedge b \in events(comp(t^+)) \Rightarrow (a, b) \in <_{t'}$
and (S8)
$\forall a, b : \mathcal{P}rogEvents \bullet (a, b) \in <_{P_M} \wedge b \in events(t') \Rightarrow (a, b) \in <_{t'}$ $\qquad\qquad\qquad\square$

Hence the order of program events that is enforced by P_M is maintained by t'.

b.) $\quad \forall p : \mathcal{P}rogEvents; \ e : \mathcal{O}bjEvents \bullet (p, e) \in <_{P_M} \wedge e \in events(t') \Rightarrow (p, e) \in <_{t'}$

Proof
Case b1 : e is an invocation event
Since $<_{P_M} \Subset <_{comp(t^+)}$ (S4)
$\forall p : \mathcal{P}rogEvents; \ c : (Op \times Val) \bullet$
$\qquad (p, inv(c)) \in <_{P_M} \wedge inv(c) \in events(comp(t^+)) \Rightarrow (p, inv(c)) \in <_{comp(t^+)}$
with (ii) we have

$\forall\, p : \mathcal{P}rogEvents;\ c : (Op \times Val) \bullet (p, inv(c)) \in <_{P_M} \land inv(c) \in events(comp(t^+)) \Rightarrow (p, inv(c)) \in <_{t'}$
and (S6)
$\forall\, p : \mathcal{P}rogEvents;\ c : (Op \times Val) \bullet (p, inv(c)) \in <_{P_M} \land inv(c) \in events(t') \Rightarrow (p, inv(c)) \in <_{t'}$

Case b2 : e is a response event
Let $inv(a, in)$ and $res(a, out)$ be in $events(P)$, and assume that $res(a, out) \in events(t')$.
Since $t' \in Trace$ (S10), $inv(a, in) \in events(t')$ (Axiom 2), and from (*Case b1*)
$(p, inv(a, in)) \in <_{P_M} \Rightarrow (p, inv(a, in)) \in <_{t'}$
and since $inv(a, in)$ must occur in t' before $res(a, out)$ (Axiom 2)
$(p, inv(a, in)) \in <_{P_M} \Rightarrow (p, res(a, out)) \in <_{t'}$.
Hence, with Axiom 4 we have
$(p, res(a, out)) \in <_{P_M} \Rightarrow (p, res(a, out)) \in <_{t'}$
and therefore
$\forall\, p : \mathcal{P}rogEvents;\ c : (Op \times Val) \bullet (p, res(c)) \in <_{P_M} \land res(c) \in events(t') \Rightarrow (p, res(c)) \in <_{t'}$

Case b3 : e is an effect event
Following the same line of reasoning as in *Case b2* (with Axiom 5 in place of Axiom 4) we have
$\forall\, p : \mathcal{P}rogEvents;\ c : (Op \times Val)\} \bullet (p, eff(c)) \in <_{P_M} \land eff(c) \in events(t') \Rightarrow (p, eff(c)) \in <_{t'}$

Hence, with (*Case b1*), (*Case b2*) and (*Case b3*) it follows that the order of a program event followed by an object event that is enforced by P_M is maintained in t', i.e.,
$\forall\, p : \mathcal{P}rogEvents;\ e : \mathcal{O}bjEvents \bullet (p, e) \in <_{P_M} \land e \in events(t') \Rightarrow (p, e) \in <_{t'}$ $\square$

c.) $\forall\, p : \mathcal{P}rogEvents;\ e : \mathcal{O}bjEvents \bullet (e, p) \in <_{P_M} \land p \in events(t') \Rightarrow (e, p) \in <_{t'}$

Proof
Case c1 : e is an invocation event
Since $<_{P_M} \Subset <_{comp(t^+)}$ (S4)
$\forall\, p : \mathcal{P}rogEvents;\ res(a, out) : events(P) \bullet$
       $(res(a, out), p) \in <_{P_M} \land p \in events(comp(t^+)) \Rightarrow (res(a, out), p) \in <_{comp(t^+)}$
and with Axiom (3)
$\forall\, p : \mathcal{P}rogEvents;\ inv(a, in), res(a, out) : events(P) \bullet$
       $(inv(a, in), p) \in <_{P_M} \land p \in events(comp(t^+)) \Rightarrow (res(a, out), p) \in <_{comp(t^+)}$
and with (S8)
$\forall\, p : \mathcal{P}rogEvents;\ inv(a, in), res(a, out) : events(P) \bullet$
       $(inv(a, in), p) \in <_{P_M} \land p \in events(t') \Rightarrow (res(a, out), p) \in <_{comp(t^+)}$.
With (iii) we have
$\forall\, p : \mathcal{P}rogEvents;\ inv(a, in), res(a, out) : events(P) \bullet$
       $(inv(a, in), p) \in <_{P_M} \land p \in events(t') \Rightarrow (res(a, out), p) \in <_{t'}$
and with Axiom (2)
$\forall\, p : \mathcal{P}rogEvents;\ inv(a, in) : events(P) \bullet$
       $(inv(a, in), p) \in <_{P_M} \land p \in events(t') \Rightarrow (inv(a, in), p) \in <_{t'}$

Case c2 : e is a response event
Since $<_{P_M} \Subset <_{comp(t^+)}$ (S4)
$\forall\, p : \mathcal{P}rogEvents;\ c : Op \times Val \bullet (res(c), p) \in <_{P_M} \land p \in events(comp(t^+)) \Rightarrow (res(c), p) \in <_{comp(t^+)}$
with (iii) we have
$\forall\, p : \mathcal{P}rogEvents;\ c : Op \times Val \bullet (res(c), p) \in <_{P_M} \land p \in events(comp(t^+)) \Rightarrow (res(c), p) \in <_{t'}$
and with (S8)
$\forall\, p : \mathcal{P}rogEvents;\ c : Op \times Val \bullet (res(c), p) \in <_{P_M} \land p \in events(t') \Rightarrow (res(c), p) \in <_{t'}$

Case c3 : e is an effect event
Following the same line of reasoning as in *Case c2* (with (iv) in place of (iii)) we have
$\forall\, p : \mathcal{P}rogEvents;\ c : Op \times Val \bullet (eff(c), p) \in <_{P_M} \land p \in events(t') \Rightarrow (eff(c), p) \in <_{t'}$

Hence, with (*Case c*1), (*Case c*2) and (*Case c*3) it follows that the order of an object event followed by a program event that is enforced by P_M is maintained in t', i.e.,
$\forall\, p : \mathcal{P}rogEvents;\ e : \mathcal{O}bjEvents \bullet\ (e, p) \in\, <_{P_M} \wedge p \in events(t') \Rightarrow (e, p) \in\, <_{t'}$ □

d.) $\forall\, a, b : \mathcal{O}bjEvents \bullet (a, b) \in\, <_{P_M} \wedge b \in events(t') \Rightarrow (a, b) \in\, <_{t'}$

Proof
*Case d*1 : a and b are events of the same operation
If the object events refer to the same operation, the reasoning is simply based on the fact that (T5) ensures that the events occur in t' in the order (i) invocation before effect, and (ii) effect before response, and no program/memory model combination can enforce an order contrary to this. (i) is common to all programs on all memory models, and (ii) can occur on any program on any memory model (by adding a fence to the operation if the memory model is not SC; under SC the effect always occurs immediately before the response).

*Case d*2 : a and b are events of different operations
If the object events refer to two different operations, we first prove that if an operation must occur before another in P on M then this is also the case in t'.
Since $<_{P_M} \subseteq\, <_{comp(t^+)}$ (S4)
$\forall\, c, d : (Op \times Val) \bullet (res(c), inv(d)) \in\, <_{P_M} \wedge inv(d) \in events(comp(t^+)) \Rightarrow (res(c), inv(d)) \in\, <_{comp(t^+)}$
with $comp(t^+)_{|ir} = comp(h^+)$ (S3) and the definition of $\prec_t$ (on page 12) we have
$\forall\, c, d : (Op \times Val) \bullet (res(c), inv(d)) \in\, <_{P_M} \wedge inv(d) \in events(comp(t^+)) \Rightarrow (res(c), inv(d)) \in \prec_{comp(h^+)}$
with $\prec_{comp(h^+)} \subseteq \prec_{h'}$ (S1) and $events(t'_{|ir}) = events(comp(t^+)_{|ir})$ (S6)
$\forall\, c, d : (Op \times Val) \bullet (res(c), inv(d)) \in\, <_{P_M} \wedge inv(d) \in events(t') \Rightarrow (res(c), inv(d)) \in \prec_{h'}$
with $t'_{|ir} = h'$ (S6) and the definition of $\prec_t$
$\forall\, c, d : (Op \times Val) \bullet (res(c), inv(d)) \in\, <_{P_M} \wedge inv(d) \in events(t') \Rightarrow (res(c), inv(d)) \in\, <_{t'}$ (∗)

Using this result, we then show that if the events of the operations must occur in a specific order in P on M then this is also the case in t'.
Let $a \in \{inv(c, in_c), res(c, out_c), eff(c, out_c)\}$ and $b \in \{inv(d, in_d), res(d, out_d), eff(d, out_d)\}$ where $c \neq d$, and assume $(a, b) \in\, <_{P_M}$. Then from Lemma 1 we have $(res(c, out_c), inv(d, in_d)) \in\, <_{P_M}$.
Hence, from (∗) we have
$(a, b) \in\, <_{P_M} \wedge inv(d, in_d) \in events(t') \Rightarrow (res(c, out_c), inv(d, in_d)) \in\, <_{t'}$
Since (T1) and (T5) ensure that if $inv(d)$ occurs in t' so do $eff(d, out_d)$ and $res(d, out_d)$, we have
$(a, b) \in\, <_{P_M} \wedge b \in events(t') \Rightarrow (res(c, out_c), inv(d, in_d)) \in\, <_{t'}$
and since (T1) and (T5) also ensure that the invocation, effect and response of an operation occur together
$(a, b) \in\, <_{P_M} \wedge b \in events(t') \Rightarrow (a, b) \in\, <_{t'}$

From (*Case d*1) and (*Case d*2) it follows that
$\forall\, a, b : \mathcal{O}bjEvents \bullet (a, b) \in\, <_{P_M} \wedge b \in events(t') \Rightarrow (a, b) \in\, <_{t'}$ □

C. Completion of matching traces

Lemma 5. For an object implementation C which is an object refinement of a specification A, and any client program P that records every invocation and response of an operation without delay, we have

$$\forall\, t : [\![P[C]]\!]_M \bullet (\exists\, t' : [\![P[A]]\!]_M \bullet t_{|global} = t'_{|global}) \Rightarrow$$
$$\exists\, t'' : [\![P[A]]\!]_M \bullet t_{|global} = t''_{|global} \wedge \exists\, t^+ : ext(t) \bullet comp(t^+)_{|ir} \sim t''_{|ir}$$

Proof
For any client program P that records every invocation and response of an operation without delay it holds that *matching* traces of $[\![P[A]]\!]_M$ and $[\![P[C]]\!]_M$, i.e., those traces for which the sequence of observable steps

is the same, will differ by at most one invocation/response pair per thread (since after such a pair there would be a further observable event before the next invocation). Hence, we consider the following 6 cases:

1. t and t' have exactly the same sequences of invocations and responses per thread. In this case, the lemma is satisfied by choosing $t'' = t'$ and $t^+ = t$.

2. t' has extra invocation/response pairs (note that abstract traces cannot be extended by just an invocation (see Axiom (7)). In this case, we choose t'' by removing the invocation/response pairs. This is always possible since the $[\![P[A]]\!]_M$ is prefix-closed (and there is at most one invocation/response pair on each thread). Then with $t^+ = t$, the lemma is satisfied.

3. t has extra invocation/response pairs. Since all responses are recorded by P, we know that there is an extension of t in $[\![P[C]]\!]_M$ with observable program steps recording each of the extra pairs. From the antecedent, there must be a matching trace in $[\![P[A]]\!]$ and hence we know that there exists a t'' which extends t' with the invocation/response pairs of t. Given this t'', the lemma is satisfied with $t^+ = t$.

4. t and t' both have invocation/response pairs, but they are different. We use the prefix of t' which does not have the invocation/response pairs (as in case 2), and extend it with the invocation/response pairs of t (as in case 3) to get t''. Then the lemma is satisfied with $t^+ = t$.

5. t has one or more extra invocations. In this case, for each invocation with an effect we add a response in t^+ and we extend t' with the matching invocation/response pairs (as in case 4) to get t''. The remaining invocations will be removed by the function $comp$.

6. t has one or more extra invocations, and t' has extra invocation/response pairs. In this case, we use the prefix of t' which does not have the extra invocation/response pairs (as in case 2), and extend t and this prefix of t' as in case 5.

$\square$