Correctness of Concurrent Objects under Weak Memory Models

Graeme Smith Kirsten Winter
Robert J. Colvin
School of Information Technology and Electrical Engineering,
The University of Queensland, Australia
smith@itee.uq.edu.au kirsten@itee.uq.edu.au r.colvin@uq.edu.au

In this paper we develop a theory for correctness of concurrent objects under weak memory models. Central to our definitions is the concept of *observations* which determine when effects of operations become visible, and hence determine the semantics of objects, under a given memory model. The resulting notion of correctness, called *object refinement*, is generic as it is parameterised by the memory model under consideration. Our theory enforces the minimal constraints on the placing of observations and on the semantics of objects that underlie object refinement. Object refinement is suitable as a reference for correctness when proving new proof methods for objects under weak memory models to be sound and complete.

1 Introduction

Linearizability [15] is widely accepted as the standard correctness criterion for concurrent objects, i.e., objects designed to be accessed simultaneously by multiple threads [16]. Recent work [5, 13, 7, 20, 8, 10] has begun examining the applicability of linearizability in the context of weak memory models of modern multicore architectures [17, 18, 2, 12, 6]. These memory models improve hardware efficiency by limiting accesses to global memory. Individual threads 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.

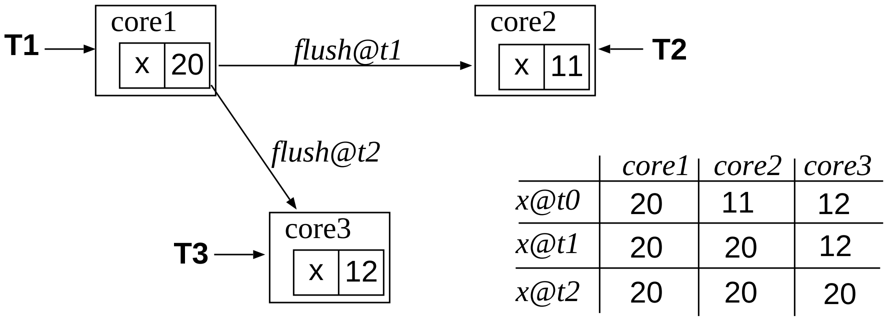
For example, on the TSO (Total Store Order) architecture [17, 18] a thread updating a global variable x stores the new value in a per-core FIFO buffer. Threads executing on that core will then read x from the buffer, rather than the global memory, until the new value is flushed from the buffer to global memory 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.

The situation is even more complex for weaker architectures, like ARM and Power [2, 12, 6]. Writes to variables are, like on TSO, local to the core on which they occur and are made available to other cores by the hardware. However, they are not necessarily made available in FIFO order. A write to variable x may be made available globally after a write to variable y although the write to x occurs before the write to y in program order.

Additionally, values of global variables are flushed at different times to different cores (this is referred to as *non-multi-copy atomicity*). This is illustrated in Figure 1 where three different cores are depicted with one thread running on each and referring to a global variable x. Assume at time $t0$ thread T1 performs a write to x, which then gets flushed to core2 at time $t1$, and later at time $t2$ to core3. The

[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 cause the code to be less efficient.

J. Derrick, B. Dongol and S. Reeves (Eds.):
Refinement Workshop 2018 (REFINE'18).
EPTCS 282, 2018, pp. 53–67, doi:10.4204/EPTCS.282.5



	core1	*core2*	*core3*
x@t0	20	11	12
x@t1	20	20	12
x@t2	20	20	20

Figure 1: Scenario for non-multi-copy atomicity in ARM and Power architectures

evaluation of *x* per core at these time instances is depicted in the table, which shows that for this scenario the value is consistent only at time *t*2 (for scenarios other than the depicted one, there might not be a consistent state at all).

To evaluate proposed notions of linearizability for weak memory models we need to argue their soundness and completeness. That is, we need to prove that an object implementation linearizes to its specification if and only if it is *correct* with respect to its specification. However, this requires a notion of correctness as a reference point. What does it mean for an implementation to be correct when executed on a weak memory model?

Traditionally, trace refinement provides this 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 [4, 3, 1]. 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, as the observable behaviour is the same.

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* [11] and *contextual refinement* [9] 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 core might be observable later on another. Instead events become observable immediately after their occurrence. For architectures with non-multi-copy atomicity, in which flushes occur out of write order, the definitions do not provide us with the right semantics of programs.

In this paper, we propose a general definition of correctness for concurrent objects running on *any* existing memory model. The aim is to provide a reference point for proving new notions of object correctness for weak memory models sound and complete. The result is a notion of object refinement which is parameterised by the memory model it refers to, and is therefore generic and can be instantiated for any weak memory model behaviour. Key to this result is a definition of the semantics of an object operating in the context of a calling client program under a weak memory model. In particular the semantics of an object's specification in the context of a client program under a weak memory model needs be defined in such a way that its behaviour maintains the intention of the specification, namely that operations are atomic.

The paper is structured as follows. In Section 2 we introduce the basic concepts of our theory including that of *observations* which is key to our definitions. Based on these concepts, Section 3 formalises the semantics of programs under a given memory model. The semantics of a concurrent object in the context of a client, 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 can then define our notion of *object refinement* under memory model M which delivers the notion of correctness for objects. Section 6, illustrates how our definition can be used to prove correctness of a case study. The paper concludes with a discussion of related work in Section 7.

2 Client programs and observation events

To investigate the behaviour of concurrent objects under weak memory models, and relate their implementation to their specification 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, spawning multiple threads T_i on multiple cores, and is affected by the memory model of the architecture it is running on. For some finite n, we have[2]

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

Following other work on concurrent objects [15, 11, 9], 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, *observation events*.

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

Operation events abstract the effects 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.

All program steps and operations are deterministic; non-determinism in a program results from the interleaving of events on different threads varying between executions.

2.1 Observation events

Central to our definitions is the notion of an additional type of event called an *observation event*. Such an event denotes the point in an execution where a program step or the response of an operation can be deemed to have been observed by *all* threads T_i of client P. Such observed events are either

- program steps that write to global program variables, or
- responses of operations that write to object variables that are shared by threads accessing the object.

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

Note that generally operation responses as such are not observable, as the returned value (if any) is written to a thread-local variable only. A subsequent program step is required to write it to a global program variable [11, 19]. Also operation internal steps and their effects are only observable on the object level but not on the program level (due to object encapsulation). Program refinement only takes into account observable program steps, and hence it is sufficient to consider only those. However, the observation of operations become essential when defining the semantics of objects in a context of a client running under weak memory model, as we will see in Section 4.

Introducing observation events allows us to decouple the occurrence of an event and its observation which might not fall together under weak memory models. The semantics specific to the memory model under consideration determines the possible placings of observation events. For example, the operational semantics for TSO given in [17], or for ARM and Power (e.g., [6]), can produce all possible executions from which we can deduce the points of observations according to the following rules: Generally, an observation will occur when *all* threads can either

(a) access a new value of a global program variable written by a program step,

(b) access the values of *all* shared object variables written by an operation, *after* the response of the operation has occurred, or

(c) when the response of a *covert* operation, whose steps or outcomes are not directly visible, has occurred.

Covert operations (referred to in case (c)) are those which do not write to any shared object variables and either have no return value or have a return value which is not written to a global program variable ([11] does not consider these operations), and as such they are not directly observable. However, from the observation of an event that must have occurred later than the response of the covert operation, we can deduce that the covert operation must have occurred, and it is thus indirectly observable. To ensure those two observations are ordered correspondingly, the observation of a covert operation is placed directly after its response. Examples of cases (a) and (b) are simply the flushes of global program and shared object variables (of which we consider only the flush to the last core when non-multi-copy atomicity is present). If an operation writes to multiple shared object variables, the observation of the last write is considered. Note that observations of writes in an operation (as in case (b)) or covert operation responses (as in case (c)) can only occur after the response of the operation.

2.2 Examples

The effects of concurrency (interleaving the behaviour of threads) as well as the effects of weak memory models leads to a partial ordering of the events of a program to which all possible executions of that program must adhere.

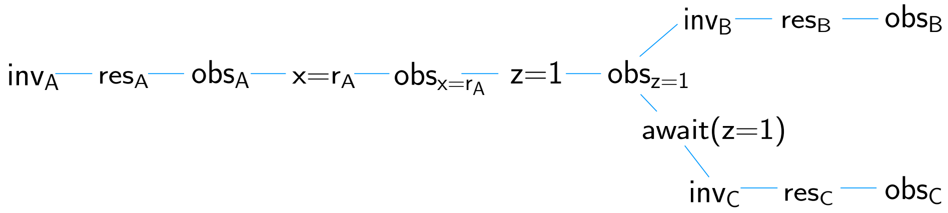
```
T1                       T2

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

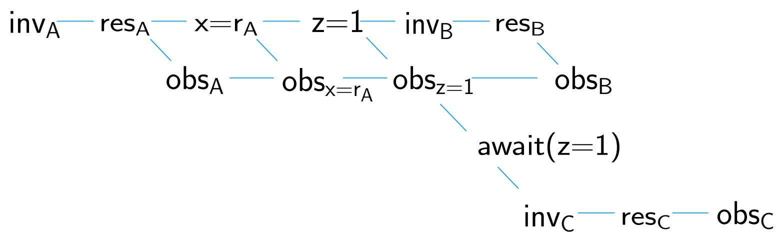
Figure 2: Client program

Consider the example of a client program given in Figure 2 with two global program variables x and z, running on two threads T1 and T2. The threads call three operations of the same object. Note that we assume that the result of any operation call is implicitly stored in a local register (e.g., $\mathsf{r_A}$ for the operation call object.A), and $\mathsf{inv_A}$ denotes the invocation of object.A, etc.

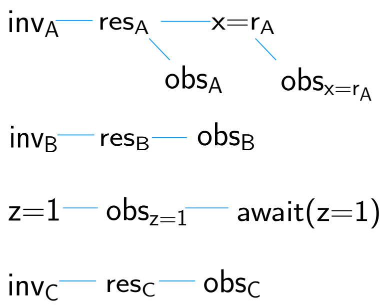
On an SC architecture (i.e., one without a weak memory model), writes to global variables occur instantaneously. Hence, the observation event for a program step occurs immediately after the program step, and that of an operation immediately after its response. The partial order of the events of the program of Figure 2 on SC is shown below.



On TSO, writes to global variables and shared object variables become available to threads on other cores when they are flushed. Furthermore, flushes occur in the same order as the writes occurred. Hence, in the partial order of the events on TSO (depicted below), $\mathsf{obs_A}$ occurs before $\mathsf{obs_{x=rA}}$ which occurs before $\mathsf{obs_{z=1}}$ which occurs before $\mathsf{obs_B}$. In contrast, the observations $\mathsf{obs_B}$ and $\mathsf{obs_C}$ are un-ordered as they occur on different threads.



On ARM and Power, flushing of variables does not occur in a FIFO manner, hence the observations are un-ordered in the partial order of the program depicted below. The partial order is split into four subgraphs which are unrelated if we assume that there are no data dependencies on shared object variables between the operations. Consequently the object events can occur out of order for as long as the object events per operation adhere to the wellformedness condition of traces.[3] The assignment to x and the await statement maintain their order with respect to their immediately preceding events, the former due to a data dependency which prohibits reordering, the latter due to the synchronisation between T1 and T2.



[3] This may lead to operations on a single thread overlapping.

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 the *partial order* over those events. The partial order of events is partly enforced by P through the program text; this is referred to as the *program order*. Additionally, in particular the placement of observation events, is determined by the semantics of the memory model as outlined in Section 2. For example, under TSO the order of writes to variables is determined by the program order, and the order of observations of such writes is the same as the order of the writes (due to the FIFO nature of the store buffer). We formalise the semantics of programs as follows.

3.1 Events

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

$$Op \mathrel{\widehat{=}} T \times Call$$

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 $\perp$ meaning 'no value'. The set of all events is defined as follows where each invocation is associated with an input, and each response and observation with an output.

$$Event \mathrel{\widehat{=}} step(T,PS) \mid obs(T,PS) \mid inv(Op,Val) \mid res(Op,Val) \mid obs(Op,Val)$$

In the remainder of this paper, we refer to $step(T,PS)$ and $obs(T,PS)$ as program events, *ProgEvents*, and to $inv(Op,Val)$, $res(Op,Val)$, and $obs(Op,Val)$ as object events, *ObjEvents*.

A program P has a set of events, $events(P)$, such that for each invocation event in the events of P, $events(P)$ also contains the corresponding response and observation events. That is, each called operation can respond and be observed.

$$\begin{aligned} &\forall\, op : Op;\ in : Val \bullet inv(op,in) \in events(P) \Rightarrow \\ &\qquad \exists\, out : Val \bullet \{res(op,out), obs(op,out)\} \subseteq events(P) \end{aligned} \tag{1}$$

3.2 Traces

The semantics of a program P is described as a set of *finite* sequences of events, referred to as *traces*.[4] 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), and an invocation of an operation always occurs before the associated response, which in turn occurs before the associated observation. Similarly, a program step always occurs before its observation, if any. In the following s_i denotes the ith element of a sequence s, and $\#s$ its length.

$$\begin{aligned} Trace \mathrel{\widehat{=}} \{t : \mathrm{seq}\,Event \mid\ & (\forall\, i,j \le \#t \bullet i \neq j \Rightarrow t_i \neq t_j) \wedge \\ & \forall\, c : Op \times Val \bullet (\forall\, j \bullet t_j = res(c) \Rightarrow \exists\, i < j \bullet t_i = inv(c)) \wedge \\ & \qquad (\forall\, j \bullet t_j = obs(c) \Rightarrow \exists\, i < j \bullet t_i = res(c)) \wedge \\ & \forall\, s : T;\ p : PS \bullet (\forall\, j \bullet t_j = obs(s,p) \Rightarrow \exists\, i < j \bullet t_i = step(s,p))\} \end{aligned}$$

[4] Since we are interested in defining a notion of correctness that readily relates to linearizability, and hence only safety properties [14, 19], we do not consider infinite sequences of events in our semantics.

The events of a trace and the order on these events are defined as follows. Note that the order $<_t$ is a total order over the events in t, as a trace describes exactly one execution.

$$
\begin{aligned}
events(t) &\mathrel{\widehat{=}} \{a : Event \mid \exists\, i \bullet t_i = a\}\\
<_t &\mathrel{\widehat{=}} \{(a,b) : Event \times Event \mid \exists\, i,j \bullet i < j \land t_i = a \land t_j = b\}
\end{aligned}
$$

3.3 Partial order

The semantics of program P on memory model M is then defined as the set of traces using only events from P and whose orders adhere to the constraints prescribed by the partial order $<_{P_M}$.

$$[\![P]\!]_M \mathrel{\widehat{=}} \{t : Trace \mid events(t) \subseteq events(P) \land <_{P_M} \Subset <_t\}$$

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

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

That is, for any event b that occurs in trace t, if this event is enforced to come after another event a by $<_{P_M}$, then event a must have also occurred 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 restricted by $<_{P_M}$.

Since we are aiming at the most general description of program semantics under any memory model, we do not explicitly prescribe $<_{P_M}$ but based on the understanding of concurrent programs, concurrent objects and their interplay (as laid out in Section 2) we can 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.

In the context of this work, the order of operation events in $<_{P_M}$ are of particular interest together with the order between operation and program events. We refer to the notion of *synchronisation* (between threads) as an event on one thread that affects the occurrence of events on another thread, and hence affects the overall order of events. 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; the communication occurs only through input and output values of the objects 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 an interface between client and object.
- A client cannot enforce a flush within an operation; the operations implementation is outside the client's control.

Figure 3 gives an intuition for the possible synchronisations by means of two scenarios of threaded behaviours of a client calling two operations A and B. 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. The second scenario on the right illustrates operation A and B overlapping in which case the synchronisation between them must be an object synchronisation which is outside the control of P.

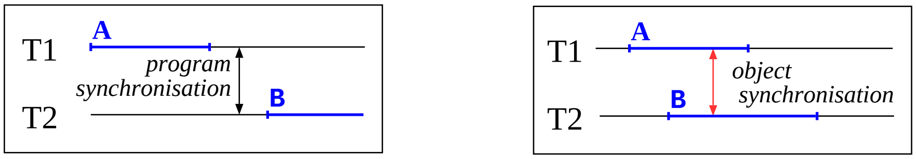


Figure 3: Scenarios of synchronisations between threads

From these assumptions we can deduce that $<_{P_M}$ can only enforce the invocation of an operation to occur *before* another event if and only if also the response is enforced to occur before the event. (The implication from right to left holds due to the wellformedness condition on traces.)

$$\forall e : Event;\ c : Op \times Val \bullet (inv(c), e) \in\ <_{P_M} \Leftrightarrow (res(c), e) \in\ <_{P_M} \tag{2}$$

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

$$\forall e : Event;\ c : Op \times Val \bullet (e, res(c)) \in\ <_{P_M} \Leftrightarrow (e, inv(c)) \in\ <_{P_M} \tag{3}$$

A similar result holds for the order of observation events with respect to program events: Due to an object's implementation, the oberservation of an operation might occur directly after its response event (e.g., if the operation is implemented with fence instructions which prevent a delay in its oberservation, or if the operation is a covert operation), which is outside the control of P. Hence $<_{P_M}$ cannot enforce the observation 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 the wellformedness condition on traces.)

$$\forall e : ProgEvents;\ c : Op \times Val \bullet (e, obs(c)) \in\ <_{P_M} \Leftrightarrow (e, inv(c)) \in\ <_{P_M} \tag{4}$$

However in some memory models, the order in which two observations can occur can be constrained (e.g., in TSO the order of observations follows the order of observed events), but only if the order of the observed events is also enforced. This means that the order of the observed operations must be enforced, and hence the response of one operation must occur before the invocation of the other.

$$\forall c, d : Op \times Val \bullet (obs(c), obs(d)) \in\ <_{P_M} \Rightarrow (res(c), inv(d)) \in\ <_{P_M}$$

A similar result also holds if $<_{P_M}$ enforces an operation observation to occur after the invoke or response of another observation. 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 observations may occur in either order) or they may overlap. For overlapping operations (necessarily executing on two different threads), $<_{P_M}$ 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.

$$\forall c, d : Op \times Val \bullet \forall e : \{inv(c), res(c), obs(c)\} \bullet (e, obs(d)) \in\ <_{P_M} \Rightarrow (res(c), inv(d)) \in\ <_{P_M} \tag{5}$$

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)

$$\forall c, d : Op \times Val \bullet \forall e_1 : \{inv(c), res(c), obs(c)\};\ e_2 : \{inv(d), res(d), obs(d)\} \bullet \\ ((e_1, e_2) \in <_{P_M} \wedge c \neq d) \ \Rightarrow\ (res(c), inv(d)) \in <_{P_M}$$

The proof constitutes the simple application of Axioms (2) to (5) to all combinations of invocation, return and observation events of two operations.

4 Semantics of objects under weak memory models

To define trace refinement between client programs using objects, we need to constrain the behaviour of a program to a particular object or collection of objects. If the program uses a collection of interacting objects, we simply consider the collection of operations (and their events) provided by all objects in the collection. Below we consider a single object only.

The semantics of an object is given as a set of *histories*, such that each history is a trace with only object events,

$$History \mathrel{\widehat{=}} \{t : Trace \mid t_{|o} = t\}$$

where $t_{|o}$ denotes the trace t restricted to only object events.

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, $[\![C]\!]_M$. Observation events are not controlled by the object and hence can occur at any time after the associated response event that the memory model allows.

For any object implementation C, $P[C]$ denotes the object C operating in program P. It is only defined when all object 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.

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

4.2 Object specification under weak memory models

An object specification A similarly has a set of object events, $events(A)$, and a prefix-closed set of histories, $[\![A]\!]$. 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 occur without interference from other operations.

To capture this in our semantics, the histories of A are restricted to those where only operations on one core are 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 4 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 the acquire which occurs second does not happen until after the response of the acquire which occurs first. On TSO and assuming T1 and T2 are

```
T1                   T2

lock.acquire;   ||   lock.acquire;
y=1             ||   z=1
```

Figure 4: Client program using lock

running on different cores, the intention is only achieved when the invocation of acquire which occurs second does not happen until after the flush of x from the acquire which occurs first. In both cases, the intention is met when the second occurrence of acquire is not invoked before the observation event of the first occurrence. In general, for any object specification to behave as intended, an operation invocation on one core does not occur before the observation event of a previously invoked operation on any other core. (The same does not need to hold for operations on the same core, as values are read locally.)

Let $core(a)$ denote the core on which an event a occurs. We assume the core can be determined from the thread of the event.

$$\begin{aligned}&\forall h : [\![A]\!] \bullet \forall c : Op \times Val;\ i \bullet h_i = inv(c) \Rightarrow \\ &\qquad \forall op : Op;\ in : Val;\ j < i \bullet h_j = inv(op,in) \wedge core(inv(op,in)) \neq core(inv(c)) \Rightarrow \\ &\qquad\qquad \exists out : Val;\ k < i \bullet h_k = obs(op,out)\end{aligned} \tag{6}$$

Additionally, since operations in a specification are intended to always respond, for each history h of A with *pending* invocations, i.e., invocations for which there is no response, the history which extends h with the missing events is also in $[\![A]\!]$.

$$\begin{aligned}&\forall h : [\![A]\!];\ c : Op \times Val \bullet \exists i \bullet h_i = inv(c) \wedge (\nexists j \bullet h_j = res(c)) \Rightarrow \\ &\qquad \exists h ^\frown hr : [\![A]\!];\ j \bullet (h ^\frown hr)_j = res(c)\end{aligned} \tag{7}$$

This is not the case for object implementations which may have operations which never return (e.g., due to an infinite loop).

Provided all object events of a program P are events of a specification A, $P[A]$ denotes the program P operating with an object whose behaviour satisfies A. The semantics of P imposes restrictions from the memory model M on to the traces and restricts the placement of observation events. The semantics of $P[A]$ is given as follows.

$$[\![P[A]]\!]_M \mathrel{\widehat{=}} \{t : [\![P]\!]_M \mid \exists h : [\![A]\!] \bullet t_{|o} = h\}$$

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 of an object. Let $t_{|global}$ denote the *observable behaviour* of a trace t, i.e., the sequence of observation events of program steps which write to global program variables.

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. We refer to this property as *weak-memory trace refinement*.

Definition 1 *Weak-memory trace refinement*

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

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*

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

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

6 Example application

6.1 Correctness on TSO

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

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

A typical concurrent implementation which is correct on SC is

```
acquire                            release         tryAcquire
  while (true) {                     x=1;            return TAS(x, 1, 0)
    if (TAS(x, 1, 0)=1) return
    while (x=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 early version of linearizability on TSO [7] proved that this implementation is also correct on TSO. However, using our definition of object refinement we can show that, in fact, it is not correct, and hence that the definition of linearizability in [7] is unsound.

Consider the client program in Figure 5, which uses a spinlock object sl, in which we assume that initially x = 1 and z = 0. One possible trace of this program is

```
    T1              T2                    T3

  z = 1;   ||   sl.acquire();   ||   await(z=1);
                sl.release();        w = sl.tryAcquire();
                y=z;
```

Figure 5: Program using spinlock

```
T1                    T2

sl.acquire();         sl.acquire();
y= y + 1;       ||    y=y + 1;
sl.release()          sl.release()
```

Figure 6: Another program using spinlock

$$\begin{aligned}&\langle \mathit{inv}(\mathsf{T2}, \mathsf{sl.acquire}), \bot), \mathit{res}((\mathsf{T2}, \mathsf{sl.acquire}, \bot), \mathit{obs}((\mathsf{T2}, \mathsf{sl.acquire}), \bot), \mathit{inv}((\mathsf{T2}, \mathsf{sl.release}), \bot),\\ &\quad \mathit{res}(\mathsf{T2}, \mathsf{sl.release}, \bot), \mathit{step}(\mathsf{T2}, \mathsf{y} = 0), \mathit{step}(\mathsf{T1}, \mathsf{z} = 1), \mathit{obs}(\mathsf{T1}, \mathsf{z} = 1), \mathit{step}(\mathsf{T3}, \mathsf{await}(\mathsf{z} = 1)),\\ &\quad \mathit{inv}((\mathsf{T3}, \mathsf{sl.tryAcquire}), \bot), \mathit{res}((\mathsf{T3}, \mathsf{sl.tryAcquire}), 0), \mathit{obs}((\mathsf{T3}, \mathsf{sl.tryAcquire}), 0), \mathit{step}(\mathsf{T3}, \mathsf{w} = 0),\\ &\quad \mathit{obs}(\mathsf{T3}, \mathsf{w} = 0), \mathit{obs}((\mathsf{T2}, \mathsf{obs.release}), \bot), \mathit{obs}(\mathsf{T2}, \mathsf{y} = 0)\rangle\end{aligned}$$

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

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

This is not an observable trace of the program running with an object satisfying the specification: if y=0 then this step 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.

The spinlock implementation without the tryAcquire operation is, however, known to be correct on TSO [18]. Again we can show this using our definition.

The traces of the implementation can be derived from the operational semantics of TSO in [17]. These 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. Hence, object refinement holds.

6.2 Correctness on ARM and Power

It is easy to show that the spinlock implementation of Section 6.1, 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 6 for which we assume that initially $\mathsf{x} = 1$ and $\mathsf{y} = 0$. Following the operational semantics of ARM and Power given in [6], one possible trace of this program is[5]

$$\begin{aligned}&\langle \mathit{inv}((\mathsf{T1}, \mathsf{acquire}), \bot), \mathit{res}((\mathsf{T1}, \mathsf{acquire}), \bot), \mathit{obs}((\mathsf{T1}, \mathsf{acquire}), \bot), \mathit{step}(\mathsf{T1}, \mathsf{y} = 1),\\ &\mathit{inv}((\mathsf{T1}, \mathsf{release}), \bot), \mathit{res}((\mathsf{T1}, \mathsf{release}), \bot), \mathit{obs}((\mathsf{T1}, \mathsf{release}), \bot), \mathit{inv}((\mathsf{T2}, \mathsf{acquire}), \bot),\\ &\mathit{res}((\mathsf{T2}, \mathsf{acquire}), \bot), \mathit{obs}((\mathsf{T2}, \mathsf{acquire}), \bot), \mathit{step}(\mathsf{T2}, \mathsf{y} = 1), \mathit{inv}((\mathsf{T2}, \mathsf{release}), \bot), \mathit{res}((\mathsf{T2}, \mathsf{release}), \bot),\\ &\mathit{obs}((\mathsf{T2}, \mathsf{release}), \bot), \mathit{obs}(\mathsf{T1}, \mathsf{y} = 1), \mathit{obs}(\mathsf{T2}, \mathsf{y} = 1)\rangle\end{aligned}$$

This trace corresponds to the response of T1's release operation being observed before its update to y.

[5] In the operational semantics of [6] the placement of observations 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.

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 $\langle obs(\mathsf{T1}, \mathsf{y} = 1), obs(\mathsf{T2}, \mathsf{y} = 1)\rangle$ of the above trace is not possible using the specification, the implementation is not correct on ARM or Power: object refinement does not hold.

7 Conclusion

In this paper, we have defined object refinement which provides a reference point for correctness of concurrent objects on any weak memory model. This allows notions of linearizability on weak memory models to be proved sound and complete.

Observational refinement [11] and contextual refinement [9] are existing notions of correctness for client/object systems on SC architectures which coincide with our notion. At the fundamental level the meaning of a program is found in how it modifies the observable state. Where we differ with those frameworks is in how interactions between the client and object are treated. In particular, to handle operations taking effect outside of their invocation/response behaviour, we introduce the notion of observation events to abstract when a change to a global variable is visible.

Doherty and Derrick [8] and Dongol et al. [10] address linearizability for weak memory models. Both papers provide arguments that their definitions of linearizability are sound. At the level of client/object systems their notions of correctness are similarly based on changes to observable variables.

In Doherty and Derrick's framework the semantics of a client operating with an abstract object specification is not constrained to behaviours in which the object's specification executes atomically. This does not affect their results since the client programs are restricted. In particular, programs like in Figure 5 which have a race between the release and tryAcquire operations are not allowed. However, our goal is to provide a general notion of correctness for *any* client program.

Dongol et al. [10] integrate the semantics of allowed reorderings of invocation and response events with the reordering semantics framework of Alglave et al. [2]. The latter is a well accepted semantics which can be specialised to TSO as well as ARM and Power. In their framework, traces of clients using specifications do not include constraints that the specifications place on the order of operations, e.g., that successive acquire operations are separated by a release operation. These constraints (captured by the specification order so) are added separately. A consequence of this construction is that correctness requires linearizability on their notion of traces and an extra condition (referred to as HB-SATISFACTION) to account for the specification order. In contrast to other work in this area, the definition of linearizability is *standard* linearizability as originally defined by Herlihy and Wing [15].

Our framework is closer to that original work on linearizability. Specifically, the allowable order of operations is included in the traces of the client program calling the specification. Therefore, unlike Dongol et al., we do not require an extra condition to handle specification order. Using our results from this paper, we would like to investigate whether standard linearizability alone can be proven sound and complete for any client program and weak memory model.

Acknowledgement This work was supported by Australian Research Council Discovery Grant DP160102457.

References

[1] M. Abadi & L. Lamport (1991): *The existence of refinement mappings*. Theoretical Computer Science 82(2), pp. 253–284, doi:10.1016/0304-3975(91)90224-P.

[2] J. Alglave, L. Maranget & M. Tautschnig (2014): *Herding Cats: Modelling, Simulation, Testing, and Data Mining for Weak Memory*. ACM Trans. Program. Lang. Syst. 36(2), pp. 7:1–7:74, doi:10.1145/2627752.

[3] R.-J.R. Back (1990): *Refinement calculus, part II: Parallel and reactive programs*. In: Stepwise Refinement of Distributed Systems Models, Formalisms, Correctness, Springer, pp. 67–93, doi:10.1007/3-540-52559-9_60.

[4] R.-J.R. Back & J. von Wright (1994): *Trace refinement of action systems*. In: *CONCUR '94*, *LNCS* 836, Springer, pp. 367–384, doi:10.1007/978-3-540-48654-1_28.

[5] S. Burckhardt, A. Gotsman, M. Musuvathi & H. Yang (2012): *Concurrent Library Correctness on the TSO Memory Model*. In H. Seidl, editor: *ESOP 2012*, *LNCS* 7211, Springer, pp. 87–107, doi:10.1007/978-3-642-28869-2_5.

[6] R.J. Colvin & G. Smith (2018): *A wide-spectrum language for verification of programs on weak memory models*. In K. Havelund, J. Peleska, B. Roscoe & E. de Vink, editors: *FM 2018*, *LNCS* 10951, Springer, pp. 240–257, doi:10.1007/978-3-319-95582-7_14.

[7] J. Derrick, G. Smith & B. Dongol (2014): *Verifying linearizability on TSO architectures*. In E. Albert & E. Sekerinski, editors: *iFM 2014*, *LNCS* 8739, Springer, pp. 341–356, doi:10.1007/978-3-319-10181-1_21.

[8] S. Doherty & J. Derrick (2016): *Linearizability and Causality*. In: *SEFM 2016*, *LNCS* 9763, Springer, pp. 45–60, doi:10.1007/978-3-319-41591-8_4.

[9] B. Dongol & L. Groves (2016): *Contextual Trace Refinement for Concurrent Objects: Safety and Progress*. In K. Ogata, M. Lawford & S. Liu, editors: *ICFEM 2016*, Springer, pp. 261–278, doi:10.1007/978-3-319-47846-3_17.

[10] B. Dongol, R. Jagadeesan, J. Riely & A. Armstrong (2018): *On abstraction and compositionality for weak-memory linearisability*. In I. Dillig & J. Palsberg, editors: *VMCAI'18*, *LNCS* 10747, Springer, pp. 183–204, doi:10.1007/978-3-319-73721-8_9.

[11] I. Filipović, P. W. O'Hearn, N. Rinetzky & H. Yang (2010): *Abstraction for concurrent objects*. Theoretical Computer Science 411(51-52), pp. 4379 – 4398, doi:10.1016/j.tcs.2010.09.021.

[12] S. Flur, K.E. Gray, C. Pulte, S. Sarkar, A. Sezgin, L. Maranget, W. Deacon & P. Sewell (2016): *Modelling the ARMv8 Architecture, Operationally: Concurrency and ISA*. In R. Bodik & R. Majumdar, editors: *POPL 2016*, ACM, pp. 608–621, doi:10.1145/2837614.2837615.

[13] A. Gotsman, M. Musuvathi & H. Yang (2012): *Show No Weakness: Sequentially Consistent Specifications of TSO Libraries*. In M. Aguilera, editor: *DISC 2012*, *LNCS* 7611, Springer, pp. 31–45, doi:10.1007/978-3-642-28869-2_5.

[14] A. Gotsman & H. Yang (2011): *Liveness-preserving atomicity abstraction*. In: *ICALP 2011*, *LNCS* 6756, Springer, pp. 453–465, doi:10.1007/978-3-642-22012-8_36.

[15] M. Herlihy & J. M. Wing (1990): *Linearizability: A Correctness Condition for Concurrent Objects*. ACM Trans. Program. Lang. Syst. 12(3), pp. 463–492, doi:10.1145/78969.78972.

[16] M. Moir & N. Shavit (2004): *Concurrent data structures*. Handbook of Data Structures and Applications, pp. 47:1–47:30, doi:10.1201/9781420035179.ch47.

[17] S. Owens, S. Sarkar & P. Sewell (2009): *A Better x86 Memory Model: x86-TSO*. In S. Berghofer, T. Nipkow, C. Urban & M. Wenzel, editors: *TPHOLs 2009*, *LNCS* 5674, Springer, pp. 391–407, doi:10.1007/978-3-642-03359-9_27.

[18] P. Sewell, S. Sarkar, S. Owens, F.Z. Nardelli & M.O. Myreen (2010): *x86-TSO: a rigorous and usable programmer's model for x86 multiprocessors*. Commun. ACM 53(7), pp. 89–97, doi:10.1145/1785414.1785443.

[19] G. Smith & K. Winter (2017): *Relating trace refinement and linearizability*. Formal Aspects of Computing 29(6), pp. 935–950, doi:10.1007/s00165-017-0418-2.

[20] O. Travkin, A. Mütze & H. Wehrheim (2013): *SPIN as a Linearizability Checker under Weak Memory Models*. In V. Bertacco & A. Legay, editors: *HVC2013*, *LNCS* 8244, Springer, pp. 311–326, doi:10.1007/978-3-319-03077-7_21.