Under consideration for publication in Formal Aspects of Computing

Relating trace refinement and linearizability

Graeme Smith and Kirsten Winter

School of Information Technology and Electrical Engineering, The University of Queensland 4072, Australia

Abstract. In the late 1980's, Back extended the notion of stepwise refinement of sequential systems to concurrent systems. By doing so he provided a definition of what it means for a concurrent system to be correct with respect to an abstract (potentially sequential) specification. This notion of refinement, referred to as *trace refinement*, was also independently proposed by Abadi and Lamport and has found widespread acceptance and application within the refinement community. Around the same time as Back's work, Herlihy and Wing proposed *linearizability* as the correctness notion for concurrent objects. Linearizability has also found widespread acceptance being regarded as the standard notion of correctness for concurrent objects in the concurrent-algorithms community.

In this paper, we provide a formal link between trace refinement and linearizability. This allows us to compare the two correctness conditions. Our comparisons show that trace refinement implies linearizability, but that linearizability does not imply trace refinement in general. However, linearizability does imply trace refinement under certain conditions. These conditions relate to (i) the fact that trace refinement can be used to prove both safety and liveness properties, whereas linearizability can only be used to prove safety properties, and (ii) the fact that trace refinement depends on the identification of when operations in the implementation are observed to occur. We discuss the consequences of these differences in the context of verifying concurrent objects.

Keywords: Trace refinement; Linearizability; Correctness; Concurrency

1. Introduction

The notion of correctness of concurrent systems in the refinement community is *trace refinement* [Bac90, BvW94, AL91]. In the concurrent-algorithms community, it is *linearizability* [HW90]. In this paper, we clarify the relationship between these two correctness conditions. Although aspects of this relationship have been addressed by a number of authors [FORY10, BEEH15, SWD14, GY11, GR14], this paper is the first, to our knowledge, to provide a complete comparison of the *standard* definitions of the conditions, and highlight their differences. Our main conclusion is that trace refinement is a stronger condition. In this paper, we discuss the consequences of the differences between the conditions in the context of verifying concurrent objects.

Correspondence and offprint requests to: Graeme Smith e-mail: smith@itee.uq.edu.au, phone: +61-7-3365 1625

Concurrent objects are objects which have been designed to allow simultaneous access by more than one thread. They include data structures and synchronisation mechanisms such as locks, and are common in modern software libraries such as `java.util.concurrent`. They may employ *coarse-grained locking*, where one thread locks the object forcing all others to wait, but for efficiency reasons are more likely to employ *fine-grained locking*, where only parts of the object are locked, e.g., two adjacent nodes in a linked list, or *non-blocking algorithms*, where no locking is employed [HS08]. A typical example of a non-blocking data structure is the Treiber stack [Tre86] the code of which is given below, where `Node` is a class with two fields `val:T` and `next:Node`, and `T_empty` is the type `T` augmented with the additional value `empty`.

```
head: Node; // shared variable
n, ss, ssn: Node; lv: T; // thread-local variables
```

```
push(v: T):
n = new(Node);
n.val = v;
repeat
    ss = head;
    n.next = ss;
until CAS(head, ss, n)
return;
```

```
pop(): T_empty
repeat
    ss = head;
    if ss = null
        return empty;
    ssn = ss.next;
    lv = ss.val;
until CAS(head, ss, ssn);
return lv;
```

A thread doing a `push` operation assigns the value being pushed onto the stack to the `val` variable of a new node stored in local variable `n`. It then repeatedly tries to make `n` the head of the stack by setting a local variable `ss` to the global variable `head`, setting `n`'s `next` variable to `ss`, and then assigning `head` to `n` provided it is still equal to `ss` (i.e., provided another thread has not in the meantime changed the value of `head`). `CAS(a,b,c)` is an atomic operation (supported by most microprocessors) which compares `a` and `b` and, if they are equal, sets `a` to `c` and returns true; otherwise it leaves `a` unchanged and returns false.

A thread doing a `pop` operation repeatedly sets `ss` to `head`, returning `empty` if `ss` is null, and otherwise setting `ssn` to `ss`'s `next` variable and local variable `lv` to `ss`'s `val` variable and, finally, assigning `ssn` to `head` and returning `lv` provided `head` is still equal to `ss`.

We would like to be able to show that this implementation satisfies a typical specification of a stack such as the following in Z [Spi92]. In such a specification, an operation has a predicate describing a relation between pre- and post-states. Intermediate states are not perceivable at this level of abstraction where implementation detail is omitted. Hence, operations correspond to atomic state transitions.

Below, S is a Z state schema. It describes the state of the object as a sequence of elements, *stack*, with each element being of a given type T. *Init* is a Z initial state schema. It extends the state schema S with a predicate describing the object's initial state, i.e., that *stack* is the empty sequence.

$$\begin{array}{|l}\multicolumn{1}{l}{S} \\ \hline stack : \mathrm{seq}\ T \\ \hline\end{array} \qquad \begin{array}{|l}\multicolumn{1}{l}{Init} \\ \hline S \\ \hline stack = \langle\,\rangle \\ \hline\end{array}$$

Push and *Pop* are Z operation schemas describing atomic state transitions. ΔS is used to introduce the variables *stack* and *stack'*, the latter denoting the value of *stack* after the operation. Variables ending in a ? denote inputs to the operation, and those ending in a ! denote operation outputs. Inputs are part of an operation's pre-state, and outputs part of its post-state.

$$\begin{array}{|l}\multicolumn{1}{l}{Push} \\ \hline \Delta S \\ v? : T \\ \hline stack' = \langle v? \rangle \frown stack \\ \hline\end{array} \qquad \begin{array}{|l}\multicolumn{1}{l}{Pop} \\ \hline \Delta S \\ lv! : T \\ \hline stack = \langle lv! \rangle \frown stack' \\ \hline\end{array}$$

The predicate of *Push* adds an element to one end of the sequence *stack*, and the predicate of *Pop* removes an element from that same end. Note that the latter predicate requires that *stack* has at least one element, the element *lv*!. In Z, the behaviour of an operation is undefined when its predicate cannot be satisfied

[Spi92]. Hence, the specification above allows *any* behaviour when *Pop* is called on an empty stack: it is up to the implementer of the specification to decide on an appropriate behaviour.

In the refinement community, the correctness of concurrent systems is provided via *trace refinement* [Bac90, BvW94, AL91]. A *trace* is a sequence of observable states of a specification or implementation.[1] Trace refinement requires that each trace of the implementation is also a trace of the abstract specification. For the stack example, the observable state is the input and output at each step. The stack's representation is not regarded as being observable, allowing it to be different in the specification and implementation (i.e., allowing for a refinement of the data structures between the levels).

To perform trace refinement, we could therefore extend the specification above with auxiliary variables, i.e., variables that do not affect the sequence of operations allowed by the specification. In the following specification, the auxiliary variables are *in* and *out* of type $T_\perp$, the type T augmented with the value $\perp$ denoting no input or output. Each schema is an extension of the similarly named schema of the original specification with additional variables and/or predicates. For the operations, the new predicates are conjoined with the original ones to describe the pre-/post-state relation of the new schema.

$$
\begin{array}{|l}
\multicolumn{1}{l}{S_{IO}} \\ \hline
S \\
in, out : T_\perp \\ \hline
\end{array}
\qquad
\begin{array}{|l}
\multicolumn{1}{l}{Init_{IO}} \\ \hline
Init \\ \hline
in = out = \perp \\ \hline
\end{array}
$$

$$
\begin{array}{|l}
\multicolumn{1}{l}{Push_{IO}} \\ \hline
Push \\
in, out : T \\ \hline
in' = v? \wedge out' = \perp \\ \hline
\end{array}
\qquad
\begin{array}{|l}
\multicolumn{1}{l}{Pop_{IO}} \\ \hline
Pop \\
in, out : T \\ \hline
in' = \perp \wedge out' = lv! \\ \hline
\end{array}
$$

Then, a possible trace, corresponding to two pushes followed by a pop, is

$$\langle in = \perp \wedge out = \perp,\ in = a \wedge out = \perp,\ in = b \wedge out = \perp,\ in = \perp \wedge out = b \rangle$$

where $a, b : T$.

The implementation needs also to be extended with auxiliary variables `in` and `out` denoting the observable inputs and outputs. The changes to these variables can be made at any step of a concrete operation but must agree with the operation's argument and return value. That is, the value of `v` must be assigned to `in` in one step of the concrete operation `push`, and for the concrete operation `pop`, the value `empty` must be assigned to `out` when the operation returns `empty` and the final value of `lv` must be assigned to `out` when it returns the value of `lv`.

In practice, such auxiliary variables are not always needed. Trace refinement is performed using simulations, and the observable state captured by the use of an abstraction relation [BvW94] or gluing invariant [Abr10]. The abstraction relation or gluing invariant relates concrete states to abstract states in such a way that state changes at the concrete level are reflected at the abstract level when they are observable.

In the concurrent-algorithms community, *linearizability* [HW90] (rather than trace refinement) is widely regarded as the notion of correctness. It compares an abstract specification of a concurrent object, where all operations are atomic and hence execute sequentially, and a concrete specification (or implementation), where operations may overlap. It requires that each operation of the concrete specification *appears* to take place atomically at some point between its invocation and return – the operation's *linearization point* [HW90] – and that the resulting sequence of such points corresponds to a sequence of operations of the abstract specification. Effectively this means that overlapping concrete operations can occur in any order in the abstract sequence, but when one concrete operation returns before another is invoked that order must be preserved in the abstract sequence.

For example, the linearization points of the Treiber stack are at line 5 of `push` when the `CAS` returns true, and at either line 7 of `pop` when `ss` is assigned `null`, or line 12 when the `CAS` returns true. Hence, if in the implementation, we begin by invoking `pop` but do not execute line 7 before invoking `push` and reaching and executing line 5, then the matching abstract history invokes *Push* first, then *Pop*.

In this paper, we provide a formal comparison of these two notions of correctness of concurrent systems.

[1] A related, event-based, notion called *action refinement* is defined in the process-algebra community [GR01].

We show that while trace refinement implies linearizability, the converse does not hold in general. We begin by providing formal definitions of trace refinement and linearizability in Sections 2 and 3, respectively. In Section 4 we show how abstract histories of operation invocations and returns can be derived from traces, and in Section 5 we formalise the relationship between concrete traces and histories. The definitions of Section 4 and 5 are then used to prove our main results in Section 6. We conclude with a discussion of these results and how they relate to the work of other authors in Section 7.

2. Trace refinement

Back [Bac90] introduced trace refinement as a way of proving the correctness of concurrent systems. A similar notion was independently proposed by Abadi and Lamport [AL91]. The basis for trace refinement is that a specification S can be represented by a set of behaviours, each behaviour being a possibly infinite sequence of states, i.e., $S \subseteq \mathrm{seq}_\infty \Sigma$, where Σ denotes the set of states of S.

A behaviour of S starts from an initial state of the specification and represents a sequence of states which the system can pass through. These states include a *global* part Σ_G (which is observable) and a *local* part Σ_L (which is not). That is, $\Sigma = \Sigma_G \times \Sigma_L$, and we use $\sigma \upharpoonright \Sigma_G$ to denote the restriction of a state σ to Σ_G.

Behaviours can be finite or infinite. Finite behaviours are those that end in a state where no further state changes are possible, i.e., there are no other behaviours of S which extend them, or behaviours which have *aborted*, i.e., are undefined after their final state. For example, a behaviour that starts with the *Pop* operation of our stack specification aborts, since its behaviour is undefined when $stack = \langle\,\rangle$.

To relate behaviours of different specifications, trace refinement focusses on the changes to the global parts of their states only. An occurrence of two consecutive states in a behaviour with the same global part is called a stuttering step. From the behaviours of S we can derive a set of traces $tr(S)$, each trace being a behaviour with all *finite* sequences of stuttering steps and the local parts of each state removed. Given $head\, s$ returns the first element of a sequence s, $tail\, s$ returns the sequence s without the first element, and $\mathrm{dom}\, s$ returns the set of indices of a sequence s, $tr(S)$ is defined formally as follows.

$$tr(S) = \{t \mid \exists\, s : S \bullet t = trace(s)\}$$

where

$$trace(s) \mathrel{\widehat=} \begin{cases} \langle\,\rangle & \text{if } s = \langle\,\rangle \\ s & \text{if } s \text{ is infinite and } \forall\, i : \mathrm{dom}\, s \bullet s(i) \upharpoonright \Sigma_G = s(i+1) \upharpoonright \Sigma_G \\ trace(tail\, s) & \text{otherwise, if } head(s) \upharpoonright \Sigma_G = head(tail(s)) \upharpoonright \Sigma_G \\ \langle head(s) \upharpoonright \Sigma_G\rangle \frown trace(tail\, s) & \text{otherwise} \end{cases}$$

The second option in the definition of $trace(s)$ ensures infinite stuttering is not removed from behaviours, whereas the third option ensures the removal of finite stuttering.

Removing stuttering steps is the key to relating behaviours of specifications with different levels of granularity, i.e., where one specification requires several steps to achieve what the other achieves in one step. The basic idea of trace refinement is that the traces of the concrete specification are a subset of the traces of the abstract specification. Additionally, however, aborting behaviours need to be considered. In Back's original definition (see Lemma 11 of [Bac90]), if the abstract specification can abort during a behaviour which starts from a given initial state σ then the concrete specification is allowed to undergo any behaviour when starting from σ. This definition was strengthened in later work by Back and von Wright [BvW94] so that the trace of the concrete behaviour starting from σ must be identical to the trace of the abstract behaviour up to the aborting state, after which the concrete specification can undergo any behaviour. In both definitions, a concrete behaviour can only abort if the trace of that behaviour is equal to the trace of an abstract behaviour which aborts.

Our definition is equivalent to that of [BvW94]. To simplify our presentation, however, we assume an aborting behaviour s is represented in the set of behaviours S by all behaviours which extend s, including a behaviour which extends s with a special "abort" state $\perp \in \Sigma$. That is, if $s \in S$ is an aborting behaviour then for all $s_1 \in seq_\infty\Sigma$, $s \frown s_1 \in S$. This assumption allows any concrete behaviours extending s to be matched to an abstract behaviour, and the behaviour $s \frown \langle\perp\rangle$ ensures that an aborting concrete behaviour must be matched to an aborting abstract behaviour.

Definition 1.[Trace refinement] A concrete specification C is a trace refinement of a specification A (written $A \sqsubseteq C$), when each trace of C is also a trace of A.

$$A \sqsubseteq C \mathrel{\widehat=} \forall\, t : tr(C) \bullet \exists\, s : tr(A) \bullet s = t \qquad \square$$

3. Linearizability

Linearizability is a correctness criterion for concurrent objects proposed by Herlihy and Wing [HW90]. It has been formalised by Derrick et al. [DSW11] among others. We base our formalisation in this section on that of Derrick and Smith [DS15] which is closer to the original definitions and terminology of Herlihy and Wing.

Linearizability is defined in terms of *histories* which are finite sequences of *events* which are invocations or responses of operations. An operation comprises the process invoking the operation from a set P, and the name of the operation from a set Op.[2]

$$Operation \mathrel{\widehat=} P \times Op$$

An occurrence of an operation also has an input from the domain In and an output from the domain Out. Both In and Out contain the value $\perp$ indicating no input or output. Such an occurrence is specified by two events: an invocation and a response, each of type $Event$. Given $op : Operation$, we let $inv(op, in)$ denote the event corresponding to op being invoked with input $in : In$, and $resp(op, out)$ denote the event corresponding to op responding with output $out : Out$. A history is a finite sequence of events, i.e., $History \mathrel{\widehat=} \operatorname{seq} Event$. For example, the following is a possible history of the Treiber stack, where p and q are processes.

$$h = \langle inv((p, \texttt{push}), 1), inv((q, \texttt{pop}), \perp), resp((p, \texttt{push}), \perp), resp((q, \texttt{pop}), 1)\rangle$$

Following [HW90], we assume each event in a history can be uniquely identified by its operation. In practice, we could annotate operation names with additional information to distinguish different occurrences of the same operation. For example, the above history h, could be extended with a second push operation as follows (where the subscript on each operation name indicates its position in the sequence of invocations, e.g., $\texttt{push}_3$ is the third operation invoked in the history).

$$\begin{aligned} h' = \langle & inv((p, \texttt{push}_1), 1), inv((q, \texttt{pop}_2), \perp), resp((p, \texttt{push}_1), \perp), resp((q, \texttt{pop}_2), 1), \\ & inv((p, \texttt{push}_3), 1), resp((p, \texttt{push}_3), \perp)\rangle \end{aligned}$$

A history h then defines a partial order $<_h$ on its operations denoting whether an operation *precedes* another. An operation op_1 precedes an operation op_2 iff op_1's response event occurs before the invocation event of op_2.

$$op_1 <_h op_2 \mathrel{\widehat=} \exists\, m, n : \operatorname{dom} h;\ in : In;\ out : Out \bullet m < n \wedge h(m) = resp(op_1, out) \wedge h(n) = inv(op_2, in)$$

In history h' above, $(p, \texttt{push}_1) <_{h'} (p, \texttt{push}_3)$ and $(q, \texttt{pop}_2) <_{h'} (p, \texttt{push}_3)$, but $(p, \texttt{push}_1)$ and $(q, \texttt{pop}_2,)$ are not related by $<_{h'}$ since they overlap.

Since operations are atomic in an abstract specification, its histories are *sequential*, i.e., each operation invocation will be followed immediately by its response. In this case, $<_h$ will be a total order. The histories of a concurrent implementation, however, may have overlapping operations and hence have the invocations and reponses of operations separated. However to be *legal*, a history should not have responses of operations for which there has not been an invocation.

$$\begin{aligned} legal(h) \mathrel{\widehat=} & \forall\, n : \operatorname{dom} h;\ op : Operation;\ out : Out \bullet \\ & \quad h(n) = resp(op, out) \Rightarrow (\exists\, m : 1 \mathrel{..} n-1;\ in : In \bullet h(m) = inv(op, in)) \end{aligned}$$

The abstract histories are also *complete*, i.e., they have a response for each invocation. This is not necessarily the case for concrete histories. For example, the subhistory of h', $\langle inv((p, \texttt{push}_1), 1), inv((q, \texttt{pop}_2), \perp)\rangle$, is also a history. Given $\#s$ is the *length* of sequence s, we define a function *complete* to remove the invocations of a history without matching responses.

[2] Herlihy and Wing also associate an object name with an operation. Here we assume this is part of the operation name.

$$complete(h) \mathrel{\widehat=} \begin{cases} \langle\,\rangle, & \text{if } h = \langle\,\rangle \\ complete(tail\,h), & \text{if } NoResp(h) \\ \langle head\,h\rangle \frown complete(tail\,h), & \text{otherwise} \end{cases}$$

where

$$\begin{aligned} NoResp(h) \mathrel{\widehat=}\ & \exists\, op : Operation;\ in : In \bullet \\ & head(h) = inv(op, in) \wedge (\nexists n : 2\,..\,\#h;\ out : Out \bullet h(n) = resp(op, out)) \end{aligned}$$

In general, to make an implementation history h into a complete history hc, we can add additional responses for those operations which have been invoked and are deemed to have occurred before removing the remaining invocations without matching responses.

$$\begin{aligned} Comp(h, hc) \mathrel{\widehat=}\ & \exists\, hr : History \bullet \\ & (\forall\, n : \operatorname{dom} hr \bullet \exists\, op : Operation;\ out : Out \bullet hr(n) = resp(op, out)) \wedge \\ & hc = complete(h \frown hr) \end{aligned}$$

Such a complete history hc is said to *match* an abstract history hs when it has the same events as hs.

$$Match(hc, hs) \mathrel{\widehat=} \forall\, ev : Event \bullet (\exists\, n : \operatorname{dom} hc \bullet hc(n) = ev) \Leftrightarrow (\exists\, m : \operatorname{dom} hs \bullet hs(m) = ev)$$

Definition 2.[Linearizability] An implementation of a concurrent object C is linearizable with respect to a specification of the object A (written $lin(C, A)$) when for each history h of C, there is a (sequential) history hs of A such that the following two conditions hold.

1. The events of a legal completion of h are identical to those of hs.
2. The precedence ordering of h is preserved by that of hs, i.e., only overlapping operations of h may be reordered with respect to each other in hs.

$$\begin{aligned} lin(C, A) \mathrel{\widehat=}\ & \forall\, h : hist(C) \bullet \\ & \exists\, hs : hist(A) \bullet (\exists\, hc : History \bullet legal(hc) \wedge Comp(h, hc) \wedge Match(hc, hs)) \wedge <_h \subseteq <_{hs} \end{aligned}$$

□

4. Deriving abstract histories

We let a specification be defined by its set of traces. Hence, we regard two specifications written using different syntax or a different specification style, but having the same traces, as being equivalent. The states in a trace of a specification are the states that can be observed before and after the operations of one of the specification's histories. Hence, a specification has one operation for every step in the abstract trace.

This means for an abstract trace s there will be $\#s - 1$ operations in the corresponding history. As we cannot derive the details of the operations, i.e., the operation name and process which invoked it, from the trace, we use the following notation to provide unique identifiers for operations: op_i^s denotes the operation responsible for the ith step in trace s, where $\#s > i$. That is, op_i^s is a label for the unknown operation responsible for the state change.

Consider the following trace of the stack corresponding to a push of $a : T$ followed by a pop.

$$s = \langle in = \bot \wedge out = \bot,\ in = a \wedge out = \bot,\ in = \bot \wedge out = a\rangle \qquad (*)$$

The corresponding abstract history will have two operations; one for each state change, $s(1)$ to $s(2)$ and $s(2)$ to $s(3)$. Given that abstract histories are sequential and complete, it will be of the form

$$\langle inv(op_1^s, a), resp(op_1^s, \bot), inv(op_2^s, \bot), resp(op_2^s, a)\rangle\,.$$

The histories of such a specification must be finite and, unlike traces, prefix-closed (where prefixes are restricted to complete histories) [HW90]. To derive histories from the set of traces of a specification, we first define the prefix-closure of the traces of a specification as follows.

$$tr\_closure(S) = \{t \mid \exists\, s : tr(S) \bullet \operatorname{dom} t \subseteq \operatorname{dom} s \wedge (\forall\, i : \operatorname{dom} t \bullet t(i) = s(i))\}$$

Next we restrict this set to finite traces as follows.

$$tr\_finite(S) = \{t \mid t \in tr\_closure(S) \land (\exists\, i : \mathbb{N} \bullet \operatorname{dom} t = 1 \mathrel{..} i)\}$$

We assume that the inputs and outputs of operations are always part of the observable state. Therefore, they can be derived from a trace. Given in_i^s and out_i^s denote the input and output to the ith step in trace s, respectively, the set of histories of a specification A are defined as follows.

Definition 3.[Abstract histories]

$$\begin{aligned} hist(A) = \{hs \mid \exists\, s : tr\_finite(A) \bullet \\ \#hs = (\#s - 1) * 2 \land \\ (\forall\, i : 1 \mathrel{..} \#s - 1 \bullet hs(i * 2 - 1) = inv(op_i^s, in_i^s) \land hs(i * 2) = resp(op_i^s, out_i^s))\} \end{aligned} \qquad \square$$

5. Relating concrete histories and traces

We let an implementation be represented by its set of traces *and* its set of histories. The traces determine the observable part of the state. For our purposes, however, the traces alone are not enough to characterise an implementation. For example, we cannot determine from the set of traces whether a state is reachable with operations occurring sequentially, or only when two or more operations overlap. Similarly, the histories alone do not allow us to distinguish implementations which only differ in their infinite behaviour.

The trace (*) above corresponds to a concrete history in which the operations which cause the two state changes have been invoked. They may have been invoked in either order, e.g., a `pop` operation may have been invoked first, but not executed line 7 before a `push` operation has been invoked and reached and executed line 5. The operations need not have returned, e.g., they only need to have reached and executed lines 5 and 12 respectively. Furthermore, other operations may also have been invoked but not reached a point where they have changed the state (and hence not returned).

Additionally, we require two constraints on the history.

1. it must be a legal history (i.e., responses cannot occur without an earlier invocation);
2. the invocation of the operation responsible for the first state change must occur before the response of the operation responsible for the second.

The second constraint is necessary since the invocation of the operation responsible for the first state change must occur before the first state change, and the response of the operation responsible for the second state change must occur at the second state change (when the return causes the state change) or any time after the second state change (when the state change is caused by an earlier step of the operation).

The above can be generalised to the following requirements on a history h corresponding to a trace t:

- the history must be legal;
- the history must include an invocation for each operation associated with a state change in the trace;

  $$Inv(h, t) \mathrel{\widehat=} \forall\, i : 1 \mathrel{..} \#t - 1 \bullet \exists\, n : \operatorname{dom} h \bullet h(n) = inv(op_i^t, in_i^t)$$

- the response of an operation associated with a state change in the trace must not occur before the invocation of an operation associated with an earlier state change;

  $$\begin{aligned} Order(h, t) \mathrel{\widehat=} \forall\, i : 2 \mathrel{..} \#t - 1;\ n : \operatorname{dom} h \bullet \\ h(n) = inv(op_{i-1}^t, in_{i-1}^t) \Rightarrow (\nexists m : 1 \mathrel{..} n - 1 \bullet h(m) = resp(op_i^t, out_i^t)) \end{aligned}$$

- if the history includes a response, it must be for an operation associated with a state change in the trace.

  $$\begin{aligned} Resp(h, t) \mathrel{\widehat=} \forall\, n : \operatorname{dom} h;\ op : Operation;\ out : Out \bullet \\ h(n) = resp(op, out) \Rightarrow (\exists\, i : 1 \mathrel{..} \#t - 1 \bullet op = op_i^t \land out = out_i^t) \end{aligned}$$

The traces and histories of an implementation C are constrained by the following two conditions.

Definition 4.[Concrete traces and histories]

(a) There is a finite trace related to each history.

$$\forall\, h : hist(C) \bullet legal(h) \land (\exists\, t : tr\_finite(C) \bullet Inv(h, t) \land Order(h, t) \land Resp(h, t))$$

(b) There is a history related to each finite trace.

$$\forall\, t : \mathit{tr\_finite}(C) \bullet \exists\, h : \mathit{hist}(C) \bullet \mathit{legal}(h) \land \mathit{Inv}(h,t) \land \mathit{Order}(h,t) \land \mathit{Resp}(h,t)$$

$\square$

6. Trace refinement and linearizability

Using the definitions of the previous sections, we now show the relationship between trace refinement and linearizability. We begin by showing that trace refinement implies linearizability.

Theorem 1. $A \sqsubseteq C \Rightarrow \mathit{lin}(C,A)$

Proof. From Definition 4(a) we know that each history h in $\mathit{hist}(C)$ is legal, and there is a trace t in $\mathit{tr\_finite}(C)$ such that h (i) has an invocation event $\mathit{inv}(op_i^t)$ for each state change in t, and (ii) has no response events other than for the invocation events of (i). That is,

$$\forall\, h : \mathit{hist}(C) \bullet \mathit{legal}(h) \land (\exists\, t : \mathit{tr\_finite}(C) \bullet \mathit{Inv}(h,t) \land \mathit{Resp}(h,t))\,.$$

Let hr be a sequence of response events for invocation events of (ii) where those response events are not in h. Let $hc = \mathit{complete}(h \frown hr)$, i.e., $h \frown hr$ with all invocations other than those of (ii) removed. hc will be legal, since h is legal and we have only added responses to the end of h for invocation events in h which did not already have a response. Hence, we have

$$\begin{array}{l}\forall\, h : \mathit{hist}(C) \bullet \mathit{legal}(h) \land \\ \quad (\exists\, t : \mathit{tr\_finite}(C) \bullet \mathit{Inv}(h,t) \land \mathit{Resp}(h,t)) \land \\ \quad (\exists\, hc : \mathit{History} \bullet \mathit{legal}(hc) \land \mathit{Comp}(h,hc))\,.\end{array}$$

hc will have one invocation and corresponding response event for each state change in t, and no other events. Hence, $\mathit{Inv}(hc,t)$. Since $A \sqsubseteq C$ (following Definition 1) there exists $s \in \mathit{tr\_finite}(A)$ such that $s = t$. Hence, $\mathit{Inv}(hc,s)$. Therefore, from Definition 3 we can deduce $\exists\, hs : \mathit{hist}(A) \bullet \mathit{Match}(hc,hs)$, i.e.,

$$\begin{array}{l}\forall\, h : \mathit{hist}(C) \bullet \mathit{legal}(h) \land \\ \quad (\exists\, t : \mathit{tr\_finite}(C) \bullet \mathit{Inv}(h,t) \land \mathit{Resp}(h,t)) \land \\ \quad (\exists\, hs : \mathit{hist}(A) \bullet \exists\, hc : \mathit{History} \bullet \mathit{legal}(hc) \land \mathit{Comp}(h,hc) \land \mathit{Match}(hc,hs))\,.\end{array}$$

Definition 4(a) also ensures that $\mathit{resp}(op_i^t, out_i^t)$ in h does not occur before $\mathit{inv}(op_{i-1}^t, in_{i-1}^t)$, for $i > 1$, i.e.,

$$\begin{array}{l}\forall\, h : \mathit{hist}(C) \bullet \mathit{legal}(h) \land \\ \quad (\exists\, t : \mathit{tr\_finite}(C) \bullet \mathit{Inv}(h,t) \land \mathit{Resp}(h,t) \land \mathit{Order}(h,t)) \land \\ \quad (\exists\, hs : \mathit{hist}(A) \bullet \exists\, hc : \mathit{History} \bullet \mathit{legal}(hc) \land \mathit{Comp}(h,hc) \land \mathit{Match}(hc,hs))\,.\end{array}$$

Hence, the operation op_{i-1}^t in h returns before the operation op_i^t or overlaps with it. Therefore, we do not have $op_i^t <_h op_{i-1}^t$, and by transitivity, we do not have $op_i^t <_h op_j^t$ for any $j < i$, and it follows that $<_h \subseteq <_{hs}$. We can conclude that

$$\forall\, h : \mathit{hist}(C) \bullet \exists\, hs : \mathit{hist}(A) \bullet (\exists\, hc : \mathit{History} \bullet \mathit{legal}(hc) \land \mathit{Comp}(h,hc) \land \mathit{Match}(hc,hs)) \land <_h \subseteq <_{hs}$$

which is the definition of $\mathit{lin}(C,A)$. $\square$

If we try to prove the converse of Theorem 1, i.e., $\mathit{lin}(C,A) \Rightarrow A \sqsubseteq C$, the proof fails. We illustrate this through five examples where $\mathit{lin}(C,A)$ holds, but $A \sqsubseteq C$ does not.

Example 1. Let A be the stack specification from Section 1. Let `push` and `pop` both be implemented in C with the following code.

```
while (true) {}
```

Any concrete history h will have a single invocation without a corresponding response. Letting $hr = \langle\,\rangle$ and $hc = \mathit{complete}(h \frown hr) = \langle\,\rangle$, we have $\mathit{legal}(hc)$ and we can choose the abstract history $hs = \langle\,\rangle$ satisfying $\mathit{Match}(h,hs)$ and $<_h \subseteq <_{hs}$. Hence, we have $\mathit{lin}(C,A)$.

However, the traces of C include infinite stuttering (recall that only finite stuttering is removed when traces are derived from behaviours), while those of A do not. Hence, we do not have $A \sqsubseteq C$. □

Example 2. Let A be the specification from Section 1, and let `push` and `pop` both be implemented in C with the following code.

```
await(head != null);
```

where `await` suspends the execution of the operation until its condition becomes true.

Assuming `head=null` initially, any concrete history will have a single invocation without a corresponding response. Again, this linearizes to the abstract history $hs = \langle\,\rangle$.

However, the traces of C include the finite trace with just an initial state, while those of A do not. So again we do not have $A \sqsubseteq C$. □

In the first example, the implementation livelocks, and in the second it deadlocks. These examples illustrate how such pathological implementations can be shown to linearize with any specification. In each example, if the specification deadlocks or livelocks, respectively, in the initial state then trace refinement would hold. However, even in this situation a problem arises when an operation in the implementation changes the state before deadlocking or livelocking. The problem is illustrated by Example 3.

Example 3. Let A be a specification which deadlocks from its intial state. Let C have a single operation with the following code (where `out:T` is a variable of the global state which is initially equal to $\perp$, and `a` is a value of `T` that is not $\perp$).

```
out = a;
await(false);
```

Any concrete history h will have a single invocation without a corresponding response, linearizing with the abstract history $hs = \langle\,\rangle$. Hence, we have $lin(C, A)$.

However, the traces of C include the finite trace with an initial state followed by the state where $out = a$. This trace is not possible in the specification. So we do not have $A \sqsubseteq C$. □

Our next example uncovers a less obvious difference between trace refinement and linearizability. It occurs when observable behaviour results from operations overlapping, but does not occur otherwise.

Example 4. Let A be the specification from Section 1, and C be the implementation from Section 1. It is well known that this implementation is linearizable with respect to the specification (see [DSW11], for example). Choose the auxiliary variable `out` to be updated in operation `pop` of C when the operation returns.

Consider the scenario where `pop` is invoked and assigns `ss` to `null` at line 7 (since the stack is empty), and then before it proceeds to line 9, `push` is invoked with argument `a` and returns. The concrete trace will be

$$\langle in = \perp \wedge out = \perp,\ in = a \wedge out = \perp,\ in = \perp \wedge out = empty\rangle$$

which is not a possible trace of the specification. Hence, we do not have $A \sqsubseteq C$. □

This example seems to suggest that we cannot prove the correctness of the Treiber stack using trace refinement. This is not the case. It *is* possible to prove its correctness using trace refinement, but only under a judicious choice of where the auxiliary variables are updated in the concrete system. This is discussed further in Section 7.

Our final example involving infinite traces is taken from Guerraoui and Ruppert [GR14].

Example 5. Let A be a *countdown object* with a single operation *op* which returns *true* or *false*. On its first occurrence, *op* nondeterministically chooses a positive integer k. The first k occurrences of *op* output *true*, and all subsequent occurrences of *op* output *false*.

In C, we have an atomic implementation of *op* which nondeterministically outputs *true* or *false*, but once it has output *false* all subsequent occurrences of *op* will also output *false*.

Each finite history of C will linearize with a history of A. Hence, $lin(C, A)$ holds. However, the infinite trace of C where *op* outputs *true* on every step, is not a trace of A. Hence, we do not have $A \sqsubseteq C$. □

We therefore show that linearizability implies trace refinement under conditions which do not allow the above examples. Specifically, given specification A and implementation C and global state Σ_G, we require that the following four constraints hold.

Constraint 1 The implementation does not introduce deadlock.

$$\forall\, t : \mathrm{seq}\, \Sigma_G \bullet t \in tr(C) \Rightarrow (\nexists t' : \mathrm{seq}_\infty\, \Sigma_G \bullet t \frown t' \in tr(A))$$

That is, if the implementation has a finite trace t then the specification should not have a trace which extends t.

The constraint that the implementation does not introduce livelock is covered by the more general constraint on infinite traces (Constraint 4) below.

Constraint 2 The implementation does not have operations which change the observable state and then deadlock or livelock.

$$\begin{array}{l} \forall\, h : hist(C);\ t : tr\_finite(C) \bullet \\ \quad legal(h) \land Inv(h, t) \land Order(h, t) \land Resp(h, t) \Rightarrow \\ \qquad (h(\#h) = inv(op^t_{\#t-1}, in^t_{\#t-1}) \Rightarrow h \frown \langle resp(op^t_{\#t-1}, out^t_{\#t-1})\rangle \in hist(C)) \end{array}$$

That is, any implementation history h corresponding to a trace t and ending with the invocation of the operation responsible for t's last state change can be extended with the response to that operation.

Constraint 3 The implementation does not introduce behaviour due to overlapping operations.

$$\begin{array}{l} \forall\, h, h' : History;\ op : Operation;\ in : In \bullet \\ \quad h \frown \langle inv(op, in)\rangle \frown h' \in hist(C) \land (\nexists j : \mathrm{dom}\, h';\ out : Out \bullet h'(j) = resp(op, out)) \Rightarrow \\ \qquad h \frown h' \in hist(C) \end{array}$$

That is, an operation *op* executing, but not returning, within a history h should not affect the outputs or order of occurrence of other operations within h.

Constraint 4 Both the specification *and* implementation are *finitely branching*, i.e., for any prefix t of a trace the number of trace prefixes which extend t by one state is finite.

$$\begin{array}{l} \forall\, t : \mathrm{seq}\, \Sigma_G;\ t' : \mathrm{seq}_\infty\, \Sigma_G \bullet \\ \quad (t \frown t' \in tr(A) \Rightarrow \{\sigma \mid \exists\, t'' : \mathrm{seq}_\infty\, \Sigma_G \bullet t \frown \langle\sigma\rangle \frown t'' \in tr(A)\} \text{ is finite}) \land \\ \quad (t \frown t' \in tr(C) \Rightarrow \{\sigma \mid \exists\, t'' : \mathrm{seq}_\infty\, \Sigma_G \bullet t \frown \langle\sigma\rangle \frown t'' \in tr(C)\} \text{ is finite}) \end{array}$$

Finite branching ensures the existence of an infinite behaviour whenever a trace can be progressively extended an infinite number of times (König's lemma [Kön90]).

Theorem 2. $lin(C, A) \Rightarrow A \sqsubseteq C$ when C does not introduce deadlock (Constraint 1), C does not have operations which change the global state before deadlocking or livelocking (Constraint 2), C does not introduce behaviour due to overlapping operations (Constraint 3), and both A and C are finitely branching (Constraint 4).

Proof. Let t be a trace in $tr\_finite(C)$. By Definition 4(b), we have

$$\forall\, t : tr\_finite(C) \bullet \exists\, h'' : hist(C) \bullet legal(h'') \land Inv(h'', t) \land Order(h'', t) \land Resp(h'', t)$$

From $Inv(h'', t)$, we know that there is an invocation in h'' for each operation op^t_i for $i \in 1 \mathrel{..} \#t - 1$. From $Resp(h'', t)$, we know that all responses in h'' are for operations op^t_i for some $i \in 1 \mathrel{..} \#t - 1$.

Given that no operation op^t_i, for $i \in 1 \mathrel{..} \#t - 1$, can deadlock or livelock (Constraint 2), there is a history h' of C which extends h'' with a response for each such operation for which h'' does not already include a response.

$$\begin{array}{l}\forall\, t : \mathit{tr\_finite}(C) \bullet \exists\, h' : \mathit{hist}(C) \bullet \\ \quad \mathit{legal}(h') \wedge \mathit{Inv}(h', t) \wedge \mathit{Order}(h', t) \wedge \mathit{Resp}(h', t) \wedge \\ \quad \#\{n \mid \exists\, i : 1 \mathrel{..} \#t - 1 \bullet h'(n) = \mathit{resp}(op_i^t, out_i^t)\} = \#t - 1\end{array}$$

Let n be the number of invocation events in h'. Since h' is legal, $n \geq \#t-1$. That is, there are $n-(\#t-1)$ invocation events for operations other than those in $\{op_i^t \mid i \in 1 \mathrel{..} \#t - 1\}$.

Since the order of invocation of events is not constrained in an implementation, there will be another history h which does not include these $n-(\#t-1)$ invocations. Furthermore, the responses of operations in h will be identical to those in h' since overlapping operations do not introduce new behaviour (Constraint 3).

$$\begin{array}{l}\forall\, t : \mathit{tr\_finite}(C) \bullet \exists\, h : \mathit{hist}(C) \bullet \\ \quad \mathit{legal}(h) \wedge \mathit{Inv}(h, t) \wedge \mathit{Order}(h, t) \wedge \mathit{Resp}(h, t) \wedge \\ \quad \#\{n \mid \exists\, i : 1 \mathrel{..} \#t - 1 \bullet h(n) = \mathit{resp}(op_i^t, out_i^t)\} = \#t - 1 \wedge \\ \quad \#\{n \mid \exists\, i : 1 \mathrel{..} \#t - 1 \bullet h(n) = \mathit{inv}(op_i^t, in_i^t)\} = \#t - 1\end{array}$$

If $\mathit{lin}(C, A)$ holds, we know from Definition 2 that there exists a history hr comprising only response events such that $\mathit{complete}(h \frown hr)$ is legal and has the same events as a history hs of A:

$$\begin{array}{l}\forall\, t : \mathit{tr\_finite}(C) \bullet \exists\, h : \mathit{hist}(C) \bullet \\ \quad \mathit{legal}(h) \wedge \mathit{Inv}(h, t) \wedge \mathit{Order}(h, t) \wedge \mathit{Resp}(h, t) \wedge \\ \quad \#\{n \mid \exists\, i : 1 \mathrel{..} \#t - 1 \bullet h(n) = \mathit{resp}(op_i^t, out_i^t)\} = \#t - 1 \wedge \\ \quad \#\{n \mid \exists\, i : 1 \mathrel{..} \#t - 1 \bullet h(n) = \mathit{inv}(op_i^t, in_i^t)\} = \#t - 1 \wedge \\ \quad (\exists\, hs : \mathit{hist}(A) \bullet \exists\, hc : \mathit{History} \bullet \mathit{legal}(hc) \wedge \mathit{Comp}(h, hc) \wedge \mathit{Match}(hc, hs))\,.\end{array}$$

Since h already has a response for each of its $\#t - 1$ invocations, $hr = \langle\,\rangle$ and $\#hs = (\#t - 1) * 2$, i.e., there is one invocation/response pair for each state change. Hence, there exists a trace s from which hs can be derived via Definition 3 with $\#t - 1$ state changes. That is,

$$\forall\, t : \mathit{tr\_finite}(C) \bullet \exists\, s : \mathit{tr\_finite}(A) \bullet \#s = \#t\,.$$

To prove the stronger condition required by Definition 1

$$\forall\, t : \mathit{tr\_finite}(C) \bullet \exists\, s : \mathit{tr\_finite}(A) \bullet s = t \qquad (\dagger)$$

we need to show that the operation which causes each state change in h is the same as that which causes the corresponding state change in hs. Note that we already know the histories h and hs have the same operations since $\mathit{Match}(hc, hs)$ holds and $\mathit{Comp}(h, hc)$ holds with $hr = \langle\,\rangle$. We just need to show they occur in the same order.

If a response of an operation op_1 in h occurs before the invocation of another op_2, then the state change associated with op_1 in t, must occur before that associated with op_2 (since $\mathit{Order}(h, t)$ holds). Therefore, t's state changes are ordered according to $<_h$, i.e., state changes are unordered only when the corresponding operations overlap.

Similarly, if a response of an operation op_1 in h occurs before the invocation of another op_2, then the invocation/response pair of the operation op_1 in hs, must occur before that associated with op_2 (by Definition 2). That is,

$$\begin{array}{l}\forall\, t : \mathit{tr\_finite}(C) \bullet \exists\, h : \mathit{hist}(C) \bullet \\ \quad \mathit{legal}(h) \wedge \mathit{Inv}(h, t) \wedge \mathit{Order}(h, t) \wedge \mathit{Resp}(h, t) \wedge \\ \quad (\exists\, hs : \mathit{hist}(A) \bullet (\exists\, hc : \mathit{History} \bullet \mathit{legal}(hc) \wedge \mathit{Comp}(h, hc) \wedge \mathit{Match}(hc, hs)) \wedge <_h \subseteq <_{hs})\end{array}$$

and so hs's operations are ordered according to $<_h$. By Definition 3, the state changes in s occur in the same order as the operations in hs. Therefore, the order of state changes in t and s can differ only for operations which overlap in h.

Hence, if there are no overlapping operations in h the condition ($\dagger$) follows immediately. If there are overlapping operations which can occur in any order in the abstract specification, it similarly follows, i.e., we choose the hs with the same order of operations as h.

The remaining case is when there is an order ω in which the overlapping operations can linearize in the implementation which is not allowed in the specification. In this case, there are two possibilities.

1. The operations can occur in the order ω in the implementation without overlapping. In this case, since the operation which occurs first according to ω can complete before that which occurs second, and so

on, proving linearizability will fail since there is no matching abstract history with that order. Hence, we have a contradiction (since we know $lin(C, A)$ holds), and so can ignore this case.

2. The operations cannot occur in the order ω in the implementation without overlapping. Since C does not introduce additional behaviour due to overlapping operations (Constraint 3), again we have a contradiction and can ignore this case.

Hence condition (†) holds. To prove that every trace in $tr(C)$ is in $tr(A)$ we need to consider the possible infinite histories. There are two cases:

1. A finite trace t of $tr\_finite(C)$ is extended by only a finite number of (progressively longer) traces in $tr\_finite(C)$. The longest of these finite traces will be in $tr(C)$. Since C does not introduce deadlock (Constraint 1), t will similarly be extended by only a finite number of (progressively longer) traces in $tr\_finite(A)$, the longest of these being in $tr(A)$.
2. A finite trace t of $tr\_finite(C)$ is extended by an infinite number of (progressively longer) traces in $tr\_finite(C)$. According to König's lemma [Kön90], since C is finitely branching (Constraint 4), $tr(C)$ will include the infinite path corresponding to the set of (progressively longer) traces. Similarly (by Constraint 4), $tr(A)$ will include the infinite path corresponding to the set of (progressively longer) traces.

Hence, we have

$$\forall\, t : tr(C) \bullet \exists\, s : tr(A) \bullet s = t$$

which is the definition of $A \sqsubseteq C$. □

7. Discussion

7.1. Results

In this paper we have shown two main results. Firstly, that trace refinement implies linearizability (Theorem 1). The consequence of this is that trace refinement provides a sound method for proving linearizability. However, it does not provide a complete method, since secondly we have shown that linearizability does not always imply trace refinement (Theorem 2). Below we discuss the consequences of this difference when using one or the other of the correctness conditions.

7.1.1. Livelock or deadlock introduction

The most obvious cases where linearizability holds, and trace refinement does not, are when the implementation introduces deadlock or livelock. In these cases, an invoked operation in the implementation may not terminate despite always terminating in the specification. Hence, linearizability cannot be used to prove that an implementation has liveness properties; an issue which has been raised by Gotsman and Yang [GY11]. It needs to be supplemented with additional techniques to prove liveness properties. Such techniques can be found in specification notations supporting trace refinement such as action systems [BvW94] and Event-B [Abr10].

It should be noted, however, that non-blocking algorithms are not necessarily livelock-free (but are used under the assumption that livelock is probabilistically very unlikely). For example, the Linux reader-writer mechanism seqlock [BC05] facilitates fast write access by making readers loop while there is a concurrent write in progress. A typical implementation is given below.

```
int x1 = 0, x2 = 0;
int c = 0;

write(int d1,d2):
1  acquire;
2  c++;
3  x1 = d1;
4  x2 = d2;
5  c++;
6  release;
```

```
read(): int []
    int c0, d1, d2;
    do {
     do {
7       c0 = c;
8    } while(c0%2 != 0);
9    d1 = x1;
10   d2 = x2;
11  } while(c != c0);
12  return {d1,d2};
```

A process wishing to write to the shared variables `x1` and `x2` acquires a software lock and increments a counter `c`. It then proceeds to write to the variables, and finally increments `c` again before releasing the lock. The lock ensures synchronisation between writers, and the counter `c` ensures the consistency of values read by other processes. The two increments of `c` ensure that it is odd when a process is writing to the variables, and even otherwise. Hence, when a process wishes to read the shared variables, it waits in a loop until `c` is even before reading them. Also, before returning it checks that the value of `c` has not changed (i.e., another write has not begun). If it has changed, the process starts over.

One possible behaviour of this is that a thread starts a write reaching and executing line 2, but does not reach and execute line 5, while one or more readers loop indefinitely (at lines 7 and 8) waiting for the write to complete. Therefore, while seqlock is linearizable with respect to a simple reader-writer specification, it cannot be verified using trace refinement unless the implementation behaviour is constrained with fairness conditions on the scheduling of threads.

7.1.2. Livelock or deadlock after a state change

Even when the specification can deadlock or livelock, an implementation which changes the state before deadlocking or livelocking, may be linearizable although not a trace refinement. This situation arises when the changed state is never observed. In proving linearizability, the invocation of the concrete operation can be removed from the implementation history in order for it to linearize with a history of the specification. However, the state cannot be similarly removed from the implementation trace; hence trace refinement does not hold.

From a practical persepective, it could be argued that since the changed state is never observed, the implementation is never seen to behave incorrectly. Hence, linearizability, rather than trace refinement, is the more appropriate correctness notion.

7.1.3. Overlapping operations

Our Example 4, where we have behaviour which only occurs when operations overlap, shows that trace refinement can sometimes be too strong to prove correctness. There is, in fact, no program calling the stack methods that could observe the "erroneous" behaviour where the `empty` value is returned by the `pop` operation after the completion of the `push` operation. The completion of the operations called by different threads would need to be recorded via assignments to program variables, and these assignments could be interleaved in either order.

In general, to verify non-blocking algorithms using trace refinement, the verifier needs to choose the points in the implementation where the state change becomes observable to coincide with the linearization points. For example, the Treiber stack can be verified using trace refinement when the observable variable `out` is updated at line 7 (when `ss` is assigned null at that line) and at line 12 (when `head` is assigned `ssn` at that line), and `in` is updated at line 5 (when `head` is assigned `n` at that line). Under this choice of observations, the trace of Example 4 is not possible (and trace refinement holds).

Hence, the notion of linearization points is important whichever correctness approach (trace refinement or linearizability) is adopted. In this particular example, linearizations points of different operations occur at different steps of the implementation. For more complex concurrent objects, two or more linearization points may coincide (see [HSY04] for a typical example). Since the linearization point of an operation occurs between its invocation and return, a situation with two or more coinciding linearization points corresponds to a behaviour which only occurs when operations overlap. In these cases, it is not possible to find an abstraction relation (or gluing invariant) to prove trace refinement: a single non-stuttering step of the implementation would need to map to two or more non-stuttering steps in the specification (something which trace refinement does not allow). Hence, trace refinement cannot be used to prove the correctness of such concurrent objects.

7.1.4. Infinite behaviour

The other case where linearizability holds, but trace refinement does not, is when the specification and implementation have different infinite behaviours (and so the implementation has either a finite or infinite trace that the specification does not have). This difference arises because linearizability is defined in terms of finite histories, and hence ignores infinite behaviour. As we have discussed in Section 6, this difference only manifests itself when either one or both of the specification and implementation are not finitely branching.

From a practical perspective, an implementation cannot have infinite branching. Therefore differences between trace refinement and linearizability only arise when the specification has infinite branching. This means that the specification may have an infinite number of progressively longer traces, but not an infinite trace. If the implementation did include the infinite trace, it would not be possible to observe that trace in a finite time. Therefore, the implementation could be argued to be correct and, hence, again linearizability could be considered the most appropriate correctness notion.

7.2. Related work

The closest work to ours is a paper by Filipović et al. [FORY10] which claims that linearizability is equivalent to a notion of *observational refinement* defined in that paper. That is, observational refinement is claimed to be both a sound and complete method for proving linearizability.

However, observational refinement differs from trace refinement in that it excludes infinite traces and behaviour which only occurs when there are overlapping operations (as in our Example 4). Instead of using input and output variables (or an abstraction relation), the authors introduce the notion of a *client*, i.e., a program that calls the methods of the concurrent object. Observational refinement compares the behaviour of an arbitrary client program accessing the implementation (via invocation and response events) and the same client program accessing the specification.

The notion of a client program can be compared to the function *tr* of trace refinement providing the same observable state space (that of the program) for the two levels of abstraction. Additionally, like *tr*, it abstracts from the concrete sub-operations which would be stuttering steps.

The comparisons between the levels is done for all possible client programs (including those comprising a single operation) starting from an arbitrary state. Hence, it ensures each state change of a client program at the concrete level (from any state) is a possible state change at the abstract level. Putting these steps together implies that any sequence of states at the concrete level is a sequence of states at the abstract level. This is like trace refinement except that (a) infinite traces are not considered, and (b) when behaviour can only occur due to overlapping operations, the order the operations occur cannot be observed (as we now explain).

The traces, which form the semantics of programs in [FORY10], are finite (Definition 9 of [FORY10]) and each response in a trace is followed by an assignment of the returned value to a variable of the client program (Figure 5 of [FORY10]). Even at the abstract level, this assignment is not performed atomically with the invocation/response pair.

This is what makes observational refinement different to standard notions of refinement in the literature. The fact that an operation's effect is not visible immediately, but only after an additional step which changes the observable state accordingly, means that it is not possible to distinguish which of two concurrent operations returns first. Consider the following example.

Example 6. Let A be a specification which allows an operation $op1$ to output value 0 when it occurs before the first occurrence of an operation $op2$ (which outputs 1), and outputs 2 after the first occurrence of $op2$. In C, $op1$ and $op2$ are defined as follows.

```
a: int; // shared variable

op1(): int                               op2(): int
1   if (a == 0) return 0;                3   a = 1;
2   return 2;                            4   return 1;
```

In this implementation, it is possible for `op2` to be invoked and execute line 3 (setting `a` to 1) and then for `op1` to be invoked and complete before `op2` executes line 4. This results in an output of 2 occurring before an output of 1. Hence, if the observable state is updated when the operations return, we do not have $A \sqsubseteq C$.

However, if the value from the operation is not observed until after a further assignment (in both the specification and implementation), observing that a 2 has been output before observing that a 1 has been output is allowed by the specification when the following sequence of events occurs: `op2` occurs, `op1` occurs, `op1`'s output of 2 is observed by assignment of this value to a variable, `op2`'s output of 1 is observed by assignment of this value to a variable. Hence, observational refinement holds. $\square$

The main results of [FORY10] are proved for specifications with only *quiescent histories*, i.e., histories in which there are no pending invocations. In other words, the proofs only hold when all operations terminate (and hence potential deadlock and livelock are ignored). A later result in the paper extends the results to specifications with non-quiescent histories, but does so by "ignoring the non-quiescent histories in the object systems." Hence, in all cases deadlock and livelock are not considered and (since infinite traces are not considered) the results of [FORY10] agree with Theorems 1 and 2.

Overall, the claim that observational refinement is equivalent to linearizability does not contradict our results as the restrictions posed in our theorems capture the differences between trace refinement and observational refinement. In contrast to [FORY10], the aim of our work is to highlight the differences between an established notion of refinement and linearizability, instead of defining a new notion of refinement that coincides with linearizability.

Bouajjani et al. [BEEH15] prove that Filipović et al.'s notion of observational refinement does not imply linearizability when operations in specifications are non-atomic. Their counter-example has an abstract history with overlapping operations, one of which never terminates.

We, however, derive the abstract histories from the traces of the specification in such a way that operations are atomic. Our claim is that the histories we derive are those of a specification that is semantically equivalent to any other exhibiting the same traces. In doing so, we treat linearizability as a semantic relationship between specifications, and not one based on a particular syntactic representation of a system.

Also related to our work is the result of Schellhorn et al. [SWD14]. They show that a form of *weak data refinement* (proved using a combination of forwards and backwards simulation) is both sound and complete for proving linearizability. Their definitions of forward and backward simulations are similar to those defined by Back and von Wright to prove trace refinement [BvW94] but without the rule for aborting traces (which they do not allow) and the rules for handling deadlock and livelock. Furthermore, they add a history variable to both the abstract and concrete states denoting the finite history that has occurred to reach that state. Hence, they are restricted to finite traces. Importantly, they require that the observable state changes in the implementation occur at the linearization points. To handle concurrent objects where two or more linearization points coincide, they allow a non-stuttering step in the implementation to refine a sequential composition of abstract non-stuttering steps (rather than just a single abstract non-stuttering step as in trace refinement).

Acknowledgements

Thanks to Ian Hayes and Robert Colvin for their comments on an earlier draft of this paper, and to John Derrick, Brijesh Dongol and Lindsay Groves for general discussion on the topic. Special thanks to the anonymous reviewers of this paper for their detailed and insightful comments. This work was supported by Australian Research Council Discovery Grants DP160102457 and DP130102901.

References

[Abr10] J.-R. Abrial. *Modeling in Event-B: system and software engineering.* Cambridge University Press, 2010.

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

[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.

[BC05] D. Bovet and M. Cesati. *Understanding The Linux Kernel.* O'Reilly & Associates, 3rd edition, 2005.

[BEEH15] A. Bouajjani, M. Emmi, C. Enea, and J. Hamza. Tractable refinement checking for concurrent objects. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL 2015)*, pages 651–662. ACM, 2015.

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

[DS15] J. Derrick and G. Smith. A framework for correctness criteria on weak memory models. In *Intenational Symposium on Formal Methods (FM 2015)*, pages 178–194. Springer, 2015.

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

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

[GR01] R. Gorrieri and A. Rensink. Action refinement. In *Handbook of Process Algebra*, chapter 16, pages 1047–1147. Elsevier, 2001.

[GR14] R. Guerraoui and E. Ruppert. Linearizability is not always a safety property. In *Networked Systems*, pages 57–69. Springer, 2014.

[GY11] A. Gotsman and H. Yang. Liveness-preserving atomicity abstraction. In *International Colloquium on Automata, Languages and Programming (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.

[HSY04] D. Hendler, N. Shavit, and L. Yerushalmi. A scalable lock-free stack algorithm. In *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA '04)*, pages 206–215. ACM Press, 2004.

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

[Kön90] D. König. *Theory of finite and infinite graphs.* Springer, 1990.

[Spi92] J.M. Spivey. *The Z Notation: A Reference Manual.* Prentice Hall, 1992.

[SWD14] G. Schellhorn, H. Wehrheim, and J. Derrick. A sound and complete proof technique for linearizability of concurrent data structures. *ACM Trans. on Computational Logic*, 2014.

[Tre86] R.K. Treiber. Systems programming: Coping with parallelism. Technical Report RJ 5118, IBM Almaden Res. Ctr., 1986.