Brief Announcement: Concurrency-Aware Linearizability

Nir Hemed
Tel Aviv University
nirh@mail.tau.ac.il

Noam Rinetzky
Tel Aviv University
maon@cs.tau.ac.il

ABSTRACT

Linearizabilty allows to describe the behaviour of concurrent objects using sequential specifications. Unfortunately, as we show in this paper, sequential specifications cannot be used for concurrent objects whose *observable behaviour* in the presence of concurrent operations *should* be different than their behaviour in the sequential setting. As a result, such *concurrency-aware objects* do not have formal specifications, which, in turn, precludes formal verification.

In this paper we present *Concurrency Aware Linearizability* (CAL), a new correctness condition which allows to formally specify the behaviour of a certain class of concurrency-aware objects. Technically, CAL is formalized as a strict extension of linearizability, where *concurrency-aware specifications* are used instead of sequential ones. We believe that CAL can be used as a basis for modular formal verification techniques for concurrency-aware objects.

Categories and Subject Descriptors

D.1.3 [**Programming Techniques**]: Concurrent Programming; D.2.4 [**Software Engineering**]: Software/Program Verification

Keywords

Linearizability; sequential specification; concurrent specification

1. INTRODUCTION

Linearizability [4] is a property of the externally-observable behaviour of concurrent objects [4]. Intuitively, a concurrent object is linearizable if in every execution each operation seems to take effect instantaneously between its invocation and response, and the resulting sequence of (seemingly instantaneous) operations respects a given sequential specification. Unfortunately, as we show below, for some concurrent objects it is *impossible* to provide a sequential specification: their behaviour in the presence of concurrent (overlapping) operations is, and should be, *observably different* from their behaviour in the sequential setting. For these objects, which we refer to as *Concurrency-Aware Concurrent Objects (CA-objects)*, the traditional notion of linearizability is simply not expressive enough to allow for describing all desired behaviours without introducing undesired ones. As a result, CA-objects are not given a formal specification. The lack of formal specifications is problematic as it precludes formal proofs.

Concurrency-Aware Linearizabilty (CAL) is a correctness condition which addresses the aforementioned problem. CAL enables programmers to provide natural and intuitive specifications for an important class of CA-objects. Technically, CAL is an extension of linearizabilty where *Concurrency-Aware specifications* are used to describe concurrency-dependent behaviours. Sequential specifications are a special case of concurrency-aware specifications in which concurrent behaviours can be explained by sequential ones.

Running Example. `Exchanger` objects (as found, e.g., in `java.util.concurrent.Exchanger`) serve as a synchronization point at which threads can pair up and *atomically swap* elements. `Exchangers` are useful in applications such as genetic algorithms and pipeline designs, and are embedded in practice in thread-pool implementations as well as other higher-level data structures [5, 6].

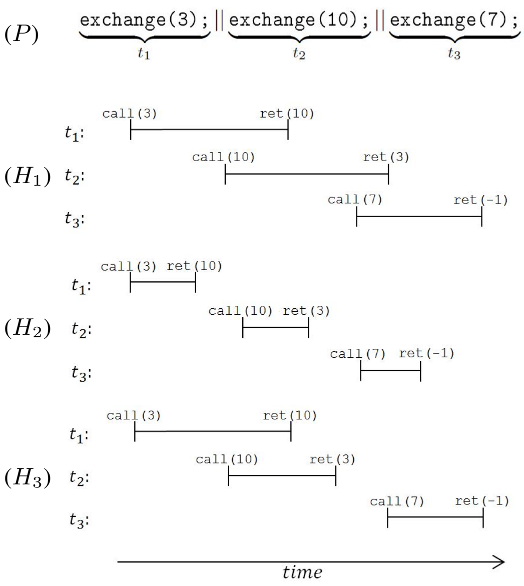
Figure 1(`E`) shows a simplified version of the wait-free exchanger of [5] in which retires are omitted. Intuitively, a client thread uses the `Exchanger` by invoking the `exchange()` method with a value that it offers to swap (in our case a positive `int`). `exchange()` attempts to find a partner thread and, if successful, instantaneously exchanges the offered value with the one offered by the partner. If a partner thread is not found, `exchange()` returns `-1`, indicating that the operation has failed. More technically, the exchange is performed by using `Offer` objects, consisting of the `data` offered for exchange and a `hole` pointer. A successful swap occurs when the `hole` pointer in the `Offer` of one thread points to the `Offer` of another thread. This can be achieved in two ways: A thread that finds that the value of `g` is null can set it to its `Offer` (line 10) and wait for a partner thread to match with (`sleep` in line 11). Upon awakening, it checks whether it was paired with another thread by executing a CAS on its own `hole` (line 12). If the CAS succeeds, then a match did not occur, and setting the `hole` pointer to point to the `fail` sentinel signals that the thread is no longer interested in the exchange. If the CAS fails then some other thread has already matched the `Offer` and the exchange can complete successfully. If `g` is not null, then the thread attempts to update the `hole` field of the `Offer` pointed to by `g` from its initial null value to its own `Offer` (line 16). An additional CAS (line 17) sets `g` back to null. By doing so, it helps to remove already-matched offers from the global pointer; hence, the CAS in line 17 is unconditional.

`Exchanger` objects do not have a formal specification. This is not surprising; describing the concurrent behaviour that requires that `exchange()` succeeds only if *two* threads invoke the method simultaneously is, as we show below, impossible using the form of sequential specifications suggested in [4]. As a result, correctness proofs of concurrent objects that utilize `Exchanger`-like ob-

PODC'14, July 15–18, 2014, Paris, France.
ACM 978-1-4503-2944-6/14/07.
http://dx.doi.org/10.1145/2611462.2611513.

(E)

```
class Offer {
  int data;
  Offer hole;
  Offer(int d){data = d; hole = null;}
}
Offer g = null;
Offer fail = new Offer(-1);
int exchange(int d){ // we assume 0 ≤ d
  Offer n = new Offer(d);
  if (CAS(g, null, n)){
    sleep(50);
    CAS(n.hole, null, fail)
    return n.hole.data;
  }
  Offer cur = g;
  bool success = CAS(cur.hole, null, n);
  CAS(g, cur, null);
  if (success)
    return cur.data;
  else return -1;
}
```



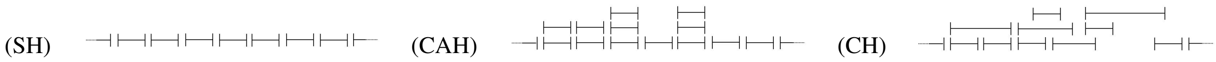


Figure 1: (E) a simplified Exchanger, (P) a client program, (H_1) a concurrent history, (H_2) an undesired sequential history, (H_3) a CA-history, a graphical depiction of a (SH) sequential history, a (CAH) CA-history, and a (CH) concurrent history.

jects are not modular. For example, the proof of the HSY-stack [3] mixes reasoning about the implementation of an (Exchanger-like) elimination array with its particular usage by the stack.

2. CONCURRENCY-AWARE LINEARIZABILITY (CAL)

Linearizability relates an implementation of a concurrent object with a sequential specification. Both the implementation and the specification are formalized as *prefix-closed sets of histories*. A history $H = \psi_1\psi_2\ldots$ is a sequence of methods invocations and responses. Specifications are given using *sequential histories* in which every response is immediately preceded by its matching invocation. Implementations, on the other hand, allow for arbitrary interleaving of actions by different threads, as long as the subsequence of actions of every thread is sequential. Informally, a concurrent object OS_C is linearizable with respect to a specification OS_A if every history H in OS_C can be *explained* by a history S in OS_A that "looks similar" to H. The similarity is formalized by a real-time relation $H \sqsubseteq_{RT} S$, which requires S to be a permutation of H preserving the per-thread order of actions and the order of non-overlapping operations.[1]

Why it is *impossible* to provide a sequential specification for Exchangers? Consider the client program P shown in Figure 1(P) which uses an Exchanger object. Figures 1(H_1-H_3) show three histories, where an exchange(n) operation returning value n' is depicted using an interval bounded by a "call(n)" and a "ret(n')" actions. Note that histories H_1 and H_3 might occur when P executes, but H_2 cannot.

History H_1 corresponds to the case where threads t_1 and t_2 exchange items 3 and 10 respectively and t_3 fails to pair-up. History H_2 is one possible sequential explanation of H_1. Using H_2 to explain H_1 raises the following problem: if H_2 is allowed by the specification then every prefix of H_2 must be allowed as-well. In particular, history H_2' in which only t_1 performs its operation should be allowed. Note that in H_2' a thread exchanges an item without finding a partner. Clearly, H_2' is an *undesired* behaviour. In fact, any sequential history that attempts to explain H_1 would allow for similar undesired behaviours. (In general, only executions in which all exchange() operations fail can be explained by sequential histories.) We conclude that any sequential specification of the Exchanger is either too restrictive or too loose.

We now turn to the definition of *concurrency-aware linearizability*. A key notion here is that of *concurrency-aware histories*. A history H is **concurrency-aware** (**CA-History**) if for any history H_1 such that $H = H_1\psi'\psi H_2$ if ψ' is a response and ψ is an invocation then the matching response of any invocation in $H_1\psi'$ is also in $H_1\psi'$. Note that a CA-history may contain concurrent operations. However, it ensures that such operations overlap pairwise. This provides the illusion that *all* concurrent operations are performed instantaneously at the same point in time. Figure 1 illustrates a sequential history (SH), a CA-history (CAH), and a concurrent history (CH). Note that every sequential history is a CA-history and every CA-history is a concurrent history, but not vice-versa. CAL extends linearizability by allowing specifications to be a (prefix-closed) set of CA-histories: A concurrent object OS_C is *CA-linearizable* with respect to a specification OS_A, if every history H in OS_C has a "similar-looking" *CA-history S* in OS_A. The "similar-looking" relation used in CAL is the same real-time order relation used to define linearizability [4]; the term *Concurrency-Aware Linearizability* emphasizes that the specification is comprised of **concurrency-aware** histories rather than a sequential ones.

Note that history H_3, depicted in Figure 1, is a *concurrency-aware history*. It describes the observable behavior as in H_1 while maintaining the same real-time order of operations and requiring

[1] For brevity, formal details, e.g., the treatment of *history completions*, are deferred to the Appendix.

that `exchange(3)` and `exchange(10)` execute concurrently and, seemingly, at the same point in time. Also note that every prefix of H_3 describes a behavior which is allowed by the implementation. Indeed, the behaviour of `Exchanger` objects can be specified precisely using CA-histories.

3. CONCLUSIONS AND FUTURE WORK

We present Concurrency-Aware Linearizabilty (CAL), a new correctness condition for an important class of CA-objects, concurrent objects whose behaviour does not have a sequential explanation. CA-objects exist in practice but currently do not have formal specifications. CAL allows providing accurate formal specifications for CA-objects using CA-histories, a restricted generalization of sequential histories. We believe that CAL can form the semantical basis for modular and reusable correctness proofs for CA-objects.

Acknowledgements. This research was supported by the EU project ADVENT.

References

[1] Abstraction for concurrent objects. *TCS*, 411(51-52), 2010.

[2] Y. Afek, M. Hakimi, and A. Morrison. Fast and scalable rendezvousing. In *Distributed Computing*. 2011.

[3] D. Hendler, N. Shavit, and L. Yerushalmi. A scalable lock-free stack algorithm. In *SPAA*, 2004.

[4] M. P. Herlihy and J. M. Wing. Linearizability: A correctness condition for concurrent objects. *TOPLAS*, 1990.

[5] W. N. Scherer III, D. Lea, and M. L. Scott. A scalable elimination-based exchange channel. *SCOOL*, 2005.

[6] W. N. Scherer III, D. Lea, and M. L. Scott. Scalable synchronous queues. In *PPoPP*, 2006.

[7] W. N. Scherer III and M. L. Scott. Nonblocking concurrent data structures with condition synchronization. In *Distributed Computing*. 2004.

APPENDIX

In this section we formalize the notion of *contentaion-aware linearizability* (CAL). We assume infinite sets of object names $o \in \mathcal{O}$, method names $f \in \mathcal{F}$, and threads identifiers $t \in \mathcal{T}$.

DEFINITION 1. *An **object action** is either an **inocation** $\psi = (t, \text{inv}\, o.f(n))$ or a **response** $\psi' = (t', \text{res}(n')o'.f')$.*

Intuitively, an invocation $\psi = (t, \text{inv}\, o.f(n))$ means transfer of control from the client to the library, and response $\psi' = (t', \text{res}(n')o'.f')$ means the return of control to the invoking client. As in [4], the observable behaviour of a concurrent object is represented by a set of histories, which are sequences of *invocations* and *responses* of methods calls.

DEFINITION 2. *A **history** H is a finite sequence of invocations and responses. We use H_i to denote the ith action of H and $H|_t$ to denote the projection of H onto actions of thread t. We denote by $\_$ an expression that is irrelevant and implicitly existentially quantified. A history is **sequential** if every response action is immediately preceded by a matching invocation. A history H is **well-formed** if invocations and responses are properly matched: for every thread t, $H|_t$ is sequential. A history is **complete** if it is well-formed and every invocation has a matching response (i.e, for every thread t, if $H|_t = \_\psi$ then ψ is a response). History H^c is a **completion** of a well-formed history H if it is complete and can be obtained from H by (possibly) extending H with some response actions and (possibly) removing some invocation actions. We denote by **complete(H)** the set of all completions of H.*

Linearizability is a relation between **object systems**, prefix-closed sets of well-formed histories. Following [4], we define it using the notion of real-time order.

DEFINITION 3. *The **real-time order** between actions of a well-formed history H is an irreflexive partial order $\prec_H$ on (indices of) object actions:*

$$
\begin{aligned}
&H_i \prec_H H_j \iff \\
&\quad \exists i \le i' < j' \le j.\, tid(H_i) = tid(H_{i'}) \wedge tid(H_j) = tid(H_{j'}) \wedge \\
&\qquad (tid(H_i) = tid(H_j) \vee H_{i'} = (\_, \text{res}\,\_) \wedge H_{j'} = (\_, \text{inv}\,\_))
\end{aligned}
$$

*A history H **agrees** with the real-time order of a history S, denoted by $H \sqsubseteq_{RT} S$, if (i) for every thread t, $H|_t = S|_t$ and (ii) there is a bijection $\pi : \{1, \cdots, |H|\} \to \{1, \cdots, |S|\}$ such that*

$$\forall i.(H_i = S_{\pi(i)}) \wedge (\forall i, j. H_i \prec_H H_j \implies S_{\pi(i)} \prec_S S_{\pi(j)}) \, .$$

Intuitively, history S "agrees" with H if in both histories every thread performs the same sequence of actions and the real-time order induced by H is a subset of that of S, i.e. $\prec_H \subseteq \prec_S$.

DEFINITION 4 (LINEARIZABILITY [4]). *Let OS_C and OS_A be object systems. We say that OS_C is **linearizable** with respect to OS_A if every history $H \in OS$ is* sequential *and*

$$\forall H \in OS_C.\, \exists H^c \in complete(H).\, \exists S \in OS_A.\, H^c \sqsubseteq_{RT} S \, .$$

We now turn on to formally define *CAL*. The key aspect is the notion of *CA-History*, which is the building block of new class of specifications which strictly extend sequential specifications. We use the notion of complete histories to provide an alternative definition for CA-histories.

DEFINITION 5 (CONCURRENCY-AWARE HISTORY). *A history H is **concurrency-aware (CA-History)** if for any history H_1 such that $H = H_1\psi'\psi H_2$ if ψ' is a response and ψ is an invocation then $H_1\psi'$ is a complete history. An object system is OS **concurrency-aware** if each $H \in OS$ is a concurrency-aware history.*

A concurrency-aware history allows for some operations to be executed concurrently by multiple threads. Moreover, it ensures that out of the set of threads that are operating concurrently, no thread will return before all other threads have invoked the operation (i.e. all operations must overlap pairwise). Figure 1 illustrates sequential history (SH), concurrency-aware history (CAH) and concurrent history (CH). Note that while every sequential history is CA, the opposite does not hold.

Extending Definition 4, Concurrency-aware linearizabilty of an object system is described using the $\sqsubseteq_{RT}$ relation to a concurrency-aware object system:

DEFINITION 6 (CONCURRENCY AWARE LINEARIZABILITY). *Let OS_C and OS_A be object systems. We say that OS_C is **concurrency-aware linearizable (CAL)** with respect to OS_A if*

$$\forall H \in OS_C.\, \exists H^c \in complete(H).\, \exists S \in OS_A.\, H^c \sqsubseteq_{RT} S$$

and every history $H \in OS_A$ is concurrency-aware.

Thus, CA-linearizable object is such that every interaction with it can be "explained" by a CA-history of some concurrency-aware object system OS_A.

Note that the same real-time order $\sqsubseteq_{RT}$ and notion of completions are used in the definitions of linearizability and concurrency-aware linearizability; the term concurrency-aware linearizability emphasizes that the specification is comprised of *concurrency-aware* histories, rather than a *sequential* ones.