Modular Verification of Concurrency-Aware Linearizability

Nir Hemed, Noam Rinetzky, Viktor Vafeiadis

Modular Verification of Concurrency-Aware Linearizability

Nir Hemed[1], Noam Rinetzky[1], and Viktor Vafeiadis[2]

[1] Tel Aviv University, Tel Aviv, Israel
[2] MPI-SWS, Germany

Abstract. Linearizability is the de facto correctness condition for concurrent objects. Informally, linearizable objects provide the illusion that each operation takes effect *instantaneously at a unique point in time* between its invocation and response. Hence, *by design*, linearizability cannot describe behaviors of *concurrency-aware concurrent objects* (CA-objects), objects in which several overlapping operations "seem to take effect *simultaneously*". In this paper, we introduce *concurrency-aware linearizability* (CAL), a generalized notion of linearizability which allows to formally describe the behavior of CA-objects. Based on CAL, we develop a thread- and procedure-modular verification technique for reasoning about CA-objects and their clients. Using our new technique, we present the first proof of linearizability of the elimination stack of Hendler *et al.* [10] in which the stack's elimination subcomponent, which is a general-purpose CA-object, is specified and verified independently of its particular usage by the stack.

1 Introduction

Linearizability [12] is a property of the externally observable behavior of concurrent objects and is considered the de facto standard for specifying concurrent objects. 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. For certain concurrent objects, however, it is *impossible* to provide a useful sequential specification: their behavior in the presence of concurrent (overlapping) operations is, and should be, *observably different* from their behavior in the sequential setting. We refer to such objects as *Concurrency-Aware Concurrent Objects (CA-objects)*. We show that the traditional notion of linearizability is not expressive enough to allow for describing all the desired behaviors of certain important CA-objects without introducing unacceptable ones, i.e., ones which their clients would find to be too lax.

Providing clear and precise specifications for concurrent objects is an important goal and is a necessary step towards developing *thread-modular compositional* verification techniques, i.e., ones which allow to reason about each thread separately (thread-modular verification) and to *compose* the proofs of concurrent objects from the proofs of their subcomponents (compositional verification). Designing such techniques is challenging because they have to take into account the possible *interference* by other threads on the shared subcomponents without exposing the internal structure of the latter.

We continue to describe the notions of CA-objects and CA-linearizability via examples. A prominent example of a CA-object is the *exchanger* object (see, e.g., `java.util.concurrent.Exchanger`). Exchangers allow threads to pair up and *atomically* swap elements so that either both threads manage to swap their elements or none of them does. Although exchangers are widely used in practice in genetic algorithms, pipeline designs, and implementations of thread pools and highly concurrent data structures such as channels, queues, and stacks [10, 21, 22, 24], they do not have a formal specification, which precludes modular proofs of their clients. This is perhaps not so surprising: exchangers are CA-objects, and as we show, they cannot be given a *useful* sequential specification (see §3). In order to specify CA-objects, we extend the notion of linearizability: we relax the requirement that specifications should be sequential, and allow them to be "*concurrency-aware*" as in the following informal exchanger specification.

$$\begin{array}{lll} \{\mathsf{true}\}\ t_1 \colon x = \texttt{exchange}(v_1) \ ||\ t_2 \colon y = \texttt{exchange}(v_2) & \{x = (\mathsf{true}, v_2) \wedge y = (\mathsf{true}, v_1)\} \\ \{\mathsf{true}\} \qquad\qquad t \colon x = \texttt{exchange}(v) & \{x = (\mathsf{false}, v)\} \end{array}$$

where the notation $t : r = \texttt{exchange(v)}$ indicates that `exchange` is invoked by thread t. This specification says that two concurrent threads t_1 and t_2 can succeed in exchanging their values but that a thread can also fail to find a partner and return back its argument.

We next consider a client of the exchanger, the *elimination stack* of Hendler et al. [10]. The elimination stack is comprised of a lock-free stack and an *elimination module* (an array of exchangers). It achieves high performance under high workloads by allowing concurrent pairs of `push` and `pop` operations to eliminate each other and thus reduce contention on the main stack. To verify the correctness of the elimination stack, one needs to ensure that every `push` operation can be eliminated by exactly one `pop` operation, and vice versa, and that the paired operations agree on the effect of the successful exchange to the observable behavior of the elimination stack as a whole. We present a reasoning technique which allows to provide natural specifications for such intricate interactions, and modularly verify their correct implementation. Intuitively, we instrument the program with an auxiliary variable that logs the sets of "*seemingly simultaneous*" operations on objects (*CA-trace*), e.g., pairs of matching successful exchange operations and singletons of failed ones.

The contributions of this paper can be summarized as follows:

- We identify the class of concurrency-aware objects in which certain operations should "seem to take effect *simultaneously*" and provide formal means to specify them using *concurrency-aware linearizability* (CAL), a generalized notion of linearizability built on top of as restricted form of concurrent specifications.
- We present a simple and effective method for verifying CAL. The unique aspects of our approach are: (i) The ability to treat a *single* atomic action as a *sequence of operations* by *different* threads which must execute completely and without interruptions, thus providing the illusion of simultaneity, and (ii) Allowing CA-objects built over other CA-objects to define their CA-trace as a function over the traces of their encapsulated objects, which makes reasoning about clients straightforward.
- We present the first *modular* proof of linearizability of the elimination stack [10] in which (i) the elimination subcomponent is verified independently of its particular usage by the stack, and (ii) the stack is verified using an implementation-independent *concurrency-aware specification* of the elimination module.

2 Motivating Examples

In this section, we describe an implementation of an exchanger object, which we use as our running example, and of one of its clients, an elimination stack. In [9], we describe another client of the exchanger, a synchronous queue [22].

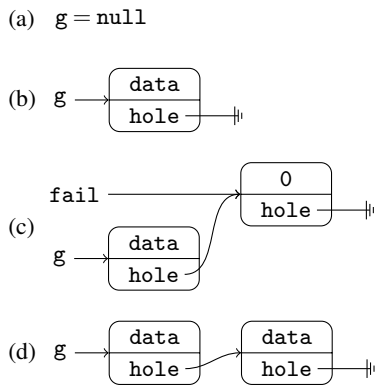
We assume an imperative programming language which allows to implement *concurrent objects* using object-local variables, dynamically (heap) allocated memory and a static (i.e., a priori fixed) number of concurrent subobjects. A program is comprised of a parallel composition of sequential commands (threads), where each thread has its own local variables. Threads share access to the dynamically allocated memory and to a static number of concurrent objects. We assume that concurrent objects follow a strict ownership discipline: (1) objects can be manipulated only by invoking their methods; (2) subobjects contained in an object o can be used only by o, the (unique) concurrent object that contains them; and (3) there is a strict separation between the parts of the memory used for the implementation of different objects. The operational semantics of our language is standard and can be found in [9]. We denote the object-local variables of an object o by $\mathsf{Vars}(o)$. For readability, we write our examples in a Java-like syntax.

2.1 Exchanger

Figure 1 shows a simplified implementation of the (wait-free) exchanger object in the `java.util.concurrent` library. A client thread uses the exchanger by invoking the `exchange` method with a value that it offers to swap. The `exchange` method attempts to find a partner thread and, if successful, instantaneously exchanges the offered value with the one offered by the partner. It then returns a pair `(true,data)`, where `data` is the partner's value of type `int`. If a partner thread is not found, `exchange` returns `(false,v)`, communicating to the client that the operation has failed. In more detail, an exchange is performed 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, as depicted in Figure 1(d).

A thread can participate in a swap in two ways. The first way happens when the thread finds that the value of `g` is null, as in the state depicted in Figure 1(a). In this case, the thread attempts to set `g` to its `Offer` (line 15) resulting in a state like the one shown in Figure 1(b). It then waits for a partner thread to match with (line 17). Upon awakening, it checks whether it was paired with another thread by executing a `CAS` on its own `hole` (line 18). 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. (The resulting state is depicted in Figure 1(c).) A failed `CAS` means that another thread has already matched the `Offer` and the exchange can complete successfully.

The second way happens when the thread finds at line 15 that `g` is not null. In this case, the thread attempts to update the `hole` field of the `Offer` pointed to by `g` from its initial null value to its own `Offer` (`CAS` at line 29). An additional `CAS` (line 31) sets `g` back to null. By doing so, the thread helps to remove an already-matched offer from the global pointer; hence, the CAS at line 31 is unconditional. Moreover, this cleanup prevents having to wait for the thread that set `g` to its offer; such a wait would compromise the wait-free property of the exchanger.




```
1  class Offer {
2    thread_id tid;
3    int data;
4    Offer hole;
5    Offer(thread_id t, int d)
6    { tid = t; data = d; hole = null; }
7  }
8  class Exchanger {
9    private Offer g = null;
10   private Offer fail = new Offer(0,0);
```

```
11 {T_E|_tid = T}
12 (bool,int) exchange(int v) {
13   Offer n = new Offer(tid,v);
14   {A}
15   if (CAS(g, null, n)){ // INIT
16     {(T_E|_tid = T ∧ n ↦ tid,v,null ∧ g = n) ∨ B(n.hole)}
17     sleep(50);
18     if (CAS(n.hole, null, fail)) // PASS
19       {T_E|_tid = T}
20       return (false,v); // FAIL
21     else {B(n.hole)}
22       return (true,n.hole.data);
23   }
24   {A}
25   Offer cur = g;
26   {A ∧ (g = cur ∨ cur.hole ≠ null)}
27   if (cur != null) {
28     {A ∧ (g = cur ∨ cur.hole ≠ null) ∧ cur ≠ null ∧ ¬s}
29     bool s = CAS(cur.hole, null, n); // XCHG
30     {(¬s ∧ A ∨ s ∧ B(cur)) ∧ cur ≠ null ∧ cur.hole ≠ null}
31     CAS(g, cur, null); // CLEAN
32     if (s) {B(cur)}
33       return (true,cur.data);
34   }
35   return (false,v); // FAIL
36 }
37 {(∃t',v'. ret = (true,v') ∧ T_E|_tid = T·E.swap(tid,v,t',v'))
   ∨ (ret = (false,v) ∧ T_E|_tid = T·(E.{(tid,ex(v) ▷ false,v)}))}
38 }
```

Fig. 1. Implementation of the exchanger CA-object annotated with its proof outline.

2.2 Elimination Stack

The *elimination stack* [10] is a scalable concurrent stack implemented using two sub-objects: a concurrent stack, `S`, which implements the internal stack data structure, and an elimination layer, `AR`. The concurrent stack, `S`, exposes `push()` and `pop()` methods that perform CAS operations to modify the `top` of the stack, and fail if there is any contention on the head of the stack. The elimination layer, `AR`, essentially acts as an exchanger object, but is implemented as an array of exchangers to reduce contention.

Figure 2 shows a simplified version of the elimination stack. A `push`ing, respectively, `pop`ping, thread first tries to perform its operation on the main stack (lines 32 and 42). If it fails due to contention, it uses the elimination layer to directly exchange a value with a concurrently executing thread: A `push`ing thread invokes `AR.exchange` (line 34) with its input value as argument, and a `pop`ping thread offers the special value `POP_SENTINAL` (line 44). When `push` calls `AR.exchange`, it randomly selects an array entry within the elimination array's range and attempts to exchange a value with another thread. The `push`ing thread checks if the return value matches the `POP_SENTINAL`. Symmetrically, a `pop`ping thread that calls `AR.exchange` checks if the return value is not `POP_SENTINAL`. Note that the `exchange` operation might fail. This might happen either because no exchange took place (the call to `exchange` returned (`false`, 0)) or because the exchange was performed between two threads executing the same operation. A thread deals with such a failure by simply retrying its operation.

```
1  class ElimArray {
2    Exchanger[] E = new Exchanger[K];
3    (bool, int) exchange(int data) {
4      int slot = random(0,K-1);
5      return E[slot].exchange(data);
6  } }
7  class Stack {
8    class Cell {int data; Cell next;}
9    Cell top = null;
10   bool push(int data) {
11     Cell h = top;
12     Cell n = new Cell(data, h);
13     return CAS(&top, h, n);
14   }
15   (bool, int) pop() {
16     Cell h = top;
17     if (h == null)
18       return (false, 0); // EMPTY
19     Cell n = h.next;
20     if (CAS(&top, h, n))
21       return (true, h.data);
22     else
23       return (false, 0);
24 } }
```

```
25 class EliminationStack {
26   final int POP_SENTINAL = INFINITY;
27   Stack S = new Stack();
28   ElimArray AR = new ElimArray();
29   bool push(int v) {
30     int d;
31     while(true) {
32       bool b = S.push(v);
33       if (b) return true;
34       (b,d) = AR.exchange(v);
35       if (d == POP_SENTINAL)
36         return true;
37   } }
38   (bool, int) pop() {
39     bool b;
40     int v;
41     while(true) {
42       (b,v) = S.pop();
43       if (b) return (true,v);
44       (b,v) = AR.exchange(POP_SENTINAL);
45       if (v != POP_SENTINAL)
46         return (true,v);
47   } }
48 }
```

Fig. 2. An implementation of the elimination stack of Hendler *et al.* [10].

3 Concurrency-Aware Linearizability (CAL)

Linearizability [12] relates (the observable behavior of) 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 method *invocation* (call) and *response* (return) actions. Specifications are given using *sequential histories*, histories in which every response is immediately preceded by its matching invocation. Implementations, on the other hand, allow 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 (execution of methods) on objects.

We claim that it is *impossible* to provide a useful sequential specification for the exchanger. Figure 3 shows a program P which uses an exchanger object and three histories, where an `exchange`(n) operation returning value n' is depicted using an interval bounded by an "inv(n)" and a "res(n')" actions. Note that histories H_1 and H_2 might occur when P executes, but H_3 cannot. Histories H_1 and H_2 correspond to the case where threads t_1 and t_2 exchange items 3 and 4, respectively, and t_3 fails to pair up. History H_3 is one possible sequential explanation of H_1. Using H_3 to explain H_1 raises the following problem: if H_3 is allowed by the specification then every prefix of H_3 must be allowed as well. In particular, history H_3' in which only t_1 performs its operation should be allowed. Note that in H_3', a thread exchanges an item without finding a partner. Clearly, H_3' is an *undesired* behavior. In fact, any sequential history that attempts to

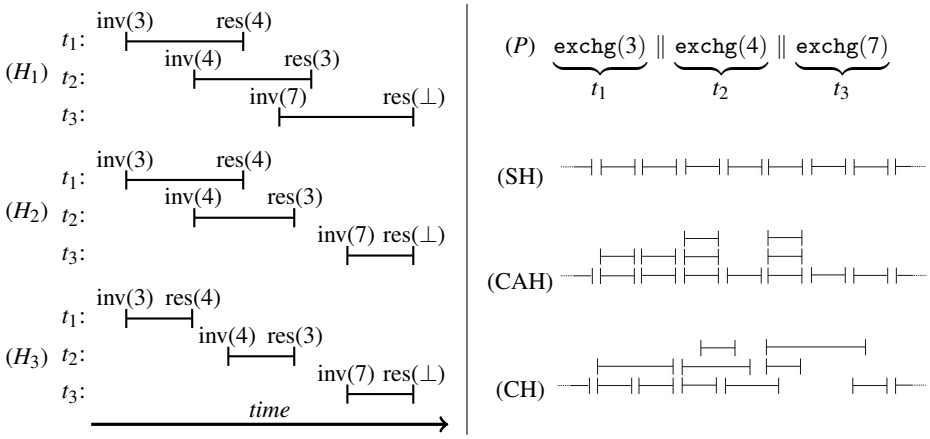


Fig. 3. A client program P together with a concurrent history (H_1), a CA-history (H_2), and an undesired sequential history (H_3). We also show schematic depictions of a sequential history (SH), a CA-history (CAH), and an arbitrary concurrent history (CH).

explain H_1 would allow for similar undesired behaviors. Indeed, sequential histories can explain only executions in which all `exchange` operations fail. We conclude that any sequential specification of the exchanger is either too restrictive or too loose.

3.1 A Formal Definition of Concurrency-Aware Linearizability

We now formalize the notion of *concurrency-aware linearizability*. We assume infinite sets of object names $o \in \mathsf{O}$, method names $f \in \mathsf{F}$, and threads identifiers $t \in \mathsf{T}$.

Definition 1. *An **object action** is either an **invocation** $\psi = (t, \mathrm{inv}\, o.f(n))$ or a **response** $\psi = (t, \mathrm{res}\, o.f \rhd n)$. We denote the* thread*,* object*, and* method *of ψ by $tid(\psi) = t$, $oid(\psi) = o$, and $fid(\psi) = f$, respectively.*

Intuitively, an invocation $\psi = (t, \mathrm{inv}\, o.f(n))$ means that thread t started executing method f on object o passing n as a parameter, and a *response* $\psi = (t, \mathrm{res}\, o.f' \rhd n)$ means that the execution of method f' terminated with a return value n.

Definition 2. *A **history** H is a finite sequence of invocations and responses. A history is **sequential** if it comprised of an alternation of invocations and responses starting with an invocation. A history H is **well-formed** if for every thread t, $H|_t$ is sequential, where $H|_t$ is the subsequence of H comprised of actions of thread t. A history is **complete** if it is well-formed and every invocation has a matching 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. An **object system** is a prefix-closed set of well-formed histories.*

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: $i \prec_H j$ if there exists $i \leq$*

$i' < j' \leq j$ *such that* $tid(H_i) = tid(H_{i'})$, $tid(H_j) = tid(H_{j'})$, $H_{i'} = (\_, \text{res}\_)$ *and* $H_{j'} = (\_, \text{inv}\_)$).

Essentially, a history records the interaction between the the client program and the object system. The interaction is recorded at the interface level of the latter at the point where control passes from the program to the object system and vice versa. Given two operations, the real-time order determines whether one operation precedes the other or whether the two are concurrent, i.e., their executions overlap.

Definition 4 (CA-traces). *An* ***operation*** *of a concurrent object o, denoted by $(t, f(n) \rhd n')$, is a pair of an invocation $(t, \text{inv}\, o.f(n))$ and its matching response $(t, \text{res}\, o.f \rhd n')$. A* ***concurrency-aware trace*** *T is a sequence of* ***CA-elements*** *where each CA-element is a pair $o.S$ of an object o and a non-empty set S of operations of o.*

Roughly speaking, every CA-element represents a set of overlapping operations on one object and a CA-trace is a sequence of such sets. CA-traces provide a uniform representation of complete histories where operations may only overlap in a pairwise manner. For example, the CA-element $o.\{(t_1, f_1(n_1) \rhd r_1), \ldots, (t_k, f_k(n_k) \rhd r_k)\}$ represents, among others, the history $((t_1, \text{inv}\, o.f_1(n_1)) \cdot \ldots \cdot (t_k, \text{inv}\, o.f_k(n_k)) \cdot\ (t_1, \text{res}\, o.f_1 \rhd r_1) \cdot \ldots \cdot (t_k, \text{res}\, o.f_k \rhd r_k))$.

Given a CA-trace T, the projection of T to a thread t, denoted by $T|_t$, is the subsequence of CA-elements of T mentioning t. Note that the projection of a trace T to thread t returns not only the operations of t but also all operations of other threads that are concurrent with some operation of thread t. Similarly, $T|_o$ denotes the subsequence of CA-elements of T mentioning o.

Let H be a complete history, and i and j indices of an invoke action $H_i = (t, \text{inv}\, o.f(n))$ and of its matching response $H_j = (t, \text{res}\, o.f \rhd n')$. The *operation* pertaining to H_i, denoted by $OP(H, i)$, is $(t, f(n) \rhd n')$. Let $J \subseteq \{1..H\}$ be a set of indices of actions in H which operate on the same object o, i.e., $\forall j \in J.\, oid(H_j) = o$. The *operation set* corresponding to J in H, denoted by $OPSet(H, J)$, is $o.\{OP(H, j) \mid j \in J\}$.

Definition 5. *A complete history H* ***agrees*** *with a CA-trace T, denoted by $H \sqsubseteq_{CAL} T$, if there is a surjective function $\pi : \{i \mid 1 \leq i \leq |H| \wedge H_i = (\_, inv\_)\} \to \{1..|T|\}$ such that*

$$\forall i, j.\, (i \prec_H j \Longrightarrow \pi(i) < \pi(j)) \wedge \forall k \in \{1, \cdots, |T|\}.\, T_k = OPSet(H, \{m \mid \pi(m) = k\})\,.$$

Intuitively, a complete history H agrees with a CA-trace T if every operation in H appears in one CA-element of T, and vice versa. Furthermore, the real-time order between the operations in H must be included in the order of the CA-elements of T that they appear in (i.e., T must preserve the order of any operations ordered according to H).

Formally, concurrency-aware linearizability of an object system is described by relating each of its histories to a corresponding CA-trace:

Definition 6 (Concurrency-Aware Linearizability). *We say that an object system, OS, is* ***concurrency-aware linearizable (CAL)*** *with respect to a set of CA-traces, $\mathcal{T}$, if $\forall H \in OS.\, \exists H^c \in complete(H).\, \exists T \in \mathcal{T}.\, H^c \sqsubseteq_{\mathsf{CAL}} T$.*

Thus, a *CA-linearizable object* is one that every interaction with it can be "explained" by a CA-trace adhering to its specification.

Note. In [8], we formalized the notion of *concurrency-aware linearizability* in terms of a relation between sets of histories. The novelty there was that the specification was comprised of *concurrency-aware* histories (CA-histories) instead of sequential ones. Informally, a CA-history allows for operations of different threads to overlap, as long as they overlap in a pairwise manner: An invoke action can follow a response action only if the latter appears at the end a complete history. As a result, a CA-histories can be seen as a sequence of sets of concurrent operations where each set is an equivalence class with respect to the real-time order. In this paper, we found that it is more convenient to work with CA-traces, which provide an equivalent alternative presentation of complete CA-histories that is insensitive to the order of actions of overlapping operations.

4 Specifying Concurrency-Aware Concurrent Objects

In this section, we gradually develop our approach for providing logical (symbolic) specifications of CA-objects by applying it to the exchanger. An accurate specification of the exchanger is one where every successful exchange corresponds to the overlapping of exactly the two operations that participated in the exchange, while an unsuccessful exchange, i.e., one that returns $(\mathsf{false}, \_)$, does not overlap with any other operation. Formally, the specification of an exchanger object `E` can be given as the set of CA-traces $S^1S^2S^3\cdots$ where each CA-element S^i is either

- $\mathtt{E}.\{(t, \mathtt{ex}(v) \rhd \mathsf{true}, v'), (t', \mathtt{ex}(v') \rhd \mathsf{true}, v)\}$ for some t, t', v, v' such that $t \neq t'$ (which in the following we will abbreviate as $\mathtt{E}.\mathit{swap}(t, v, t', v')$), or
- $\mathtt{E}.\{(t, \mathtt{ex}(v) \rhd \mathsf{false}, v)\}$ for some thread t and value v.

This specification, however, has a very global nature and is therefore cumbersome to use when reasoning about a particular exchange.

What we would like is a *local* way to specify CA-objects that is amenable to logical (syntactic) treatment. Our idea is to specify the effect of individual operations using Hoare triples [13], as is common in the sequential setting. Indeed, Herlihy and Wing [12] have also adopted this approach to describe the set of histories in the sequential specification of linearizable concurrent objects. Can we provide such a specification to the exchanger?

As a first attempt, consider the *concurrent* specification shown in §1. This specification states that *only* two threads that execute `exchange()` concurrently can match and successfully swap elements, while a thread that failed to find a partner fails to swap.

This specification may appear intuitive, but it is difficult to give it a formal meaning. The standard interpretation of Hoare triples is insufficient, because it precludes thread-modular compositional reasoning. The most obvious problem is that it is not possible to reason about the body of one thread in a sequential manner because the specification explicitly contains the parallel composition operator. A second problem is that it is difficult to adapt the concurrent specification of the exchange operations to an *agreed asymmetric view* in the context in which it is used. For example, when verifying the elimination stack, we would like to pretend that the exchange operation of the pushing thread happens right before that of the popping thread. This would allow to correctly interpret the simultaneous exchange operations as an elimination of a `push(`n`)` operation by a `pop()` which returns n.

To overcome the first problem, we extend the specification with an auxiliary variable $\mathcal{T}_{\mathtt{E}}$ recording the CA-trace witnessing that the exchanger is CAL. The specification of the exchange operation says that if initially the recorded trace was T, then after the exchange operation, it contains one more CA-element, corresponding either to the successful exchange if `exchange()` returns true or to the unsuccessful exchange otherwise.

$$\begin{array}{l}\{\mathcal{T}_{\mathtt{E}}|_{\mathtt{tid}} = T\} \quad \mathtt{tid}\colon \mathit{ret} = \mathtt{E.exchange}(v)\\ \left\{\begin{array}{l}(\exists t', v'.\ \mathit{ret} = (\mathsf{true}, v') \wedge \mathcal{T}_{\mathtt{E}}|_{\mathtt{tid}} = T \cdot \mathtt{E}.\mathit{swap}(\mathtt{tid}, v, t', v') \wedge t' \neq \mathtt{tid})\\ \vee\ (\mathit{ret} = (\mathsf{false}, v) \wedge \mathcal{T}_{\mathtt{E}}|_{\mathtt{tid}} = T \cdot (\mathtt{E}.\{(\mathtt{tid}, \mathtt{ex}(v) \rhd \mathsf{false}, v)\}))\end{array}\right\}\end{array}$$

Note that in the precondition and the postcondition, we do not describe the contents of the entire trace, but rather only of its projection to the current thread. We do so because there may be other exchanges running concurrently to the specified exchange, which may also append CA-elements to the recorded trace. To ensure that our specification is usable in a concurrent setting, we thus ensure that the precondition and postcondition are *stable* under interference from other threads, i.e., that concurrent operations cannot invalidate these assertions.

To address the second problem, we only need to perform a minor change. We do not change the specification as such, only the understanding of the auxiliary variable $\mathcal{T}_{\mathtt{E}}$. Instead of having for each object one auxiliary variable that records its CA-trace, we have one global auxiliary variable $\mathcal{T}$ that records the CA-traces for all the objects, and define $\mathcal{T}_{\mathtt{E}}$ to be the view of $\mathcal{T}$ according to object `E`. Our key idea is to let the exchanger module define $\mathcal{T}_{\mathtt{E}}$ as a function of $\mathcal{T}$. For the exchanger, we simply define $\mathcal{T}_{\mathtt{E}}$ to be the projection of $\mathcal{T}$ to the CA-elements of the exchanger (*i.e.*, $\mathcal{T}_{\mathtt{E}} = \mathcal{T}|_{\mathtt{E}}$).

Logging the object interaction using an auxiliary history variable. To specify and verify CAL, we instrument the program with an auxiliary variable $\mathcal{T}$ that records the CA-trace that is equivalent to a given concurrent history. Our idea is to add auxiliary assignments to the programs that append CA-elements to $\mathcal{T}$ at the appropriate points.

Since multiple objects can manipulate $\mathcal{T}$, the specification of an object o should not directly mention o, but rather its view on $\mathcal{T}$, which we denote as $\mathcal{T}_o$. A simple choice would be to define this view to be $\mathcal{T}|_o$, the projection of the trace to the CA-elements of object o. While this works for objects that do not depend on subobjects, it does not enable compositional verification of higher-level objects. The reason is that the desired equivalent CA-trace of a higher-level object is typically determined by the CA-traces of its subobjects. If, however, we want to verify an object compositionally, we are not allowed to peek into the implementations of its subobjects in order to add auxiliary assignments to $\mathcal{T}$.

Instead, we require for each object o to provide a function F_o from the CA-elements of its immediate subobjects to CA-traces containing only operations for o. Given such a function F_o, we define its total extension $\hat{F}_o$ as the function that given an element a returns $F_o(a)$ if this is defined or a otherwise. Note that $\hat{F}_o$ is idempotent and that for disjoint objects o and o', $\hat{F}_o \circ \hat{F}_{o'} = \hat{F}_{o'} \circ \hat{F}_o$. Next, we define $\overline{F_o}$ to recursively apply $\hat{F}_{o_i}$ for all objects o_i encapsulated by o. This is defined by induction on the object nesting depth. At each level, if o depends on objects $o_1, \ldots, o_n$, we define $\overline{F_o} \triangleq \hat{F}_o \circ (\overline{F_{o_1}} \circ \ldots \circ \overline{F_{o_n}})$.

Again, because of encapsulation, the order in which $\overline{F_{o_1}}$ to $\overline{F_{o_n}}$ are composed does not matter. Finally, define $\mathcal{T}_o \triangleq \overline{F_o}(\mathcal{T})$.

Encoding interference and cooperation using rely-guarantee conditions. Next, since the exchange operations are concurrent, we cannot merely give a sequential specification in Hoare logic, but instead use rely/guarantee reasoning [15], a more expressive formalism that allows expressing concurrent specifications. In rely/guarantee, each program C is specified not only by a precondition P and a postcondition Q, but also by a rely condition R and a guarantee condition G, which we have written as $R,G \Vdash \{P\}\ C\ \{Q\}$. These rely/guarantee conditions are parameterized by thread identifiers and describe the interaction between threads. For a thread t, the rely condition R^t records the interference that t might incur from the other threads, while the guarantee G^t records the effect t is allowed to have on other threads. Rely/guarantee gives thread-modular reasoning as it exposes the interaction between threads without referring to the code of other threads.

Internally, in the verification of the exchanger, these conditions will correlate the concrete state manipulated by the algorithm and the recorded history. For example, they require that when a thread successfully modifies the `g.hole` to point to its own offer, it also logs in $\mathcal{T}$ a CA-element which records the successful exchange (see §5).

From the client's perspective, however, the internal definitions of $R^{\mathtt{tid}}$ and $G^{\mathtt{tid}}$ are irrelevant. For them to be usable, however, they should adhere to a few minimal constraints, which are common for any object o:

- For every two distinct threads $t \neq t'$, we should have $G^t \Rightarrow R^{t'}$. This is the standard requirement in rely/guarantee reasoning ensuring that multiple methods of o may be invoked in parallel.
- The methods of o may only modify the auxiliary history variable, $\mathcal{T}$, the parts of the memory used in its own representation, and (via method calls) the state of its concurrent subobjects. Moreover, they may only append onto $\mathcal{T}$ entries corresponding to o and its encapsulated objects, and pertaining only to threads currently executing one of its methods. Formally, this is $G^t \Rightarrow (\exists T.\ \ \mathcal{T} = \overleftarrow{\mathcal{T}} \cdot T \wedge T = T|_o = T|_t \wedge \forall x \notin \{h\} \cup \mathsf{Vars}(o).\ x = \overleftarrow{x}\ )$, where we use the hook arrow notation to represent the value of a program variable in prior state.
- The object o does not assume anything about the private state of other objects, and allows them to extend the auxiliary history variable, $\mathcal{T}$. Formally, we require that $\textsc{irrelevant}_o^t \Rightarrow R^t$ where $\textsc{irrelevant}_o^t \triangleq \exists T.\ \ \mathcal{T}_o = \overleftarrow{\mathcal{T}_o} \cdot T \wedge T|_t = T|_o = \varepsilon \wedge (\forall x \in \mathsf{Vars}(o).\ x = \overleftarrow{x}\ )$.

Finally, since there are may be multiple threads running concurrently, the precondition and postcondition of the `exchange` method, we take the projection of $\mathcal{T}_{\mathtt{E}}$ to the thread of interest (i.e., $\mathcal{T}_{\mathtt{E}}|_{\mathtt{tid}}$). As is standard in Hoare logic, we use the logical variable T to record the initial value of $\mathcal{T}_{\mathtt{E}}|_{\mathtt{tid}}$.

Stack specification. The specification of the elimination stack as well as the ordinary concurrent stack it contains is expressed in a similar style. Technically, we say that a sequential history of stack operations is *well-defined* over an initial stack, if executing the (successful) operations in order is possible and yields the same results for the *pop*

operations. A history is *well-formed* with respect to the stack object, denoted $\mathrm{WF}_{\mathtt{S}}(H)$, if $H|_{\mathtt{S}}$ is a sequential well-defined history over the empty initial stack. The specifications for the stack methods $f \in \{push, pop\}$ are:

$$R^t, G^t \Vdash \{\mathrm{WF}_{\mathtt{S}}(\mathcal{T}_{\mathtt{S}}) \wedge \mathcal{T}_{\mathtt{S}}|_t = H\}\ t\colon r := \mathrm{S}.f(n)\ \{\mathrm{WF}_{\mathtt{S}}(\mathcal{T}_{\mathtt{S}}) \wedge \mathcal{T}_{\mathtt{S}}|_t = H \cdot (\mathrm{S}.\{(t, f(n) \rhd r)\})\}$$

The abstract value of a concurrent object, if needed (e.g., to determine the result of a *pop()* operation), can be "computed" by replaying the logged actions.

5 Verifying the Exchanger and the Elimination Stack

In this section, we prove that the elimination stack is linearizable by verifying each of objects—the exchanger, the elimination array, the central stack, and the elimination stack—modularly. For space reasons, we only present the key ingredients of the proof. The full proofs can be found in [9].

We start with the elimination array, whose correctness is the simplest to demonstrate. The elimination array, `AR`, encapsulates an array of exchanger objects `E[0]`, ..., `E[K-1]` and exposes the same specification as a single exchanger. To verify that it conforms to its specification, we define the $F_{\mathtt{AR}}$ function as $F_{\mathtt{AR}}(\mathtt{E}[i].\mathcal{S}) \triangleq (\mathtt{AR}.\mathcal{S})$, i.e., an exchange done by any of `AR`'s exchanger subobjects is converted to look like an exchange on the elimination array. This hides the implementation of the elimination array from its clients, in our case, the elimination stack. To verify the implementation of the elimination array, we pick the rely condition to be the conjunction of all the rely conditions of the encapsulated objects, $R^t_{\mathtt{AR}} \triangleq \bigwedge_i R_{\mathtt{E}[i]}$, and the guarantee condition to be the disjunction of the corresponding guarantee conditions, $G^t_{\mathtt{AR}} \triangleq \bigvee_i G^t_{\mathtt{E}[i]}$. The postcondition of `AR.exchange` follows directly from the postcondition of `E[slot].exchange` by observing that $h_{\mathtt{AR}} = \overline{F_{\mathtt{AR}}}(h_{\mathtt{E[slot]}})$.

Verifying that the central stack is a straightforward proof of linearizability, and we omit it for brevity. Next, we consider the elimination stack assuming that the central stack, `S`, and the elimination array, `AR`, satisfy their specifications. Given our setup, this proof is also straightforward. The key step is to define the function $F_{\mathtt{ES}}$ correctly:

$$\begin{gathered}
F_{\mathtt{ES}}((\mathrm{S}.(t, \mathtt{push}(n) \rhd \mathsf{true}))) \triangleq ((\mathtt{ES}.(t, \mathtt{push}(n) \rhd \mathsf{true}))) \\
F_{\mathtt{ES}}((\mathrm{S}.(t, \mathtt{pop}() \rhd \mathsf{true}, n))) \triangleq ((\mathtt{ES}.(t, \mathtt{pop}() \rhd \mathsf{true}, n))) \\
F_{\mathtt{ES}}\left(\mathtt{AR}.\left\{\begin{array}{l}(t, ex(n) \rhd \mathsf{true}, \infty),\\ (t', ex(\infty) \rhd \mathsf{true}, n)\end{array}\right\}\right) \triangleq \begin{array}{l}(\mathtt{ES}.(t, \mathtt{push}(n) \rhd \mathsf{true})) \cdot \\ (\mathtt{ES}.(t', \mathtt{pop}() \rhd \mathsf{true}, n))\end{array} \quad \text{provided } n \neq \infty \\
F_{\mathtt{ES}}(\mathrm{S}._) \triangleq \varepsilon \qquad F_{\mathtt{ES}}(\mathtt{AR}._) \triangleq \varepsilon
\end{gathered}$$

This function picks as linearization points the successful pushes and pops of `S`, as well as a successful exchange where the exchanged values are ∞ and $n \neq \infty$. In the latter case, the push is linearized before the pop. All other operations are ignored.

5.1 Verifying the Exchanger

We move on to the verification of the exchanger, which is more challenging than that of its clients. As the exchanger does not encapsulate other objects besides memory cells,

$$
\begin{aligned}
\text{INIT}^t &\triangleq [\exists n.\ \overleftarrow{g} = \mathit{null} \wedge n.\mathsf{tid} = t \wedge n.\mathsf{hole} = \mathit{null} \wedge g = n]_g \\
\text{CLEAN}^t &\triangleq [\overleftarrow{g}.\mathsf{hole} \neq \mathit{null} \wedge g' = \mathit{null}]_g \\
\text{PASS}^t &\triangleq [g.\overleftarrow{\mathsf{hole}} = \mathit{null} \wedge g.\mathit{tid} = t \wedge g.\mathsf{hole} = \mathit{fail}]_{g.\mathsf{hole}} \\
\text{XCHG}^t &\triangleq \begin{bmatrix} \exists n \neq \mathit{fail}.\ n.\mathsf{tid} = t \wedge g.\overleftarrow{\mathsf{hole}} = \mathit{null} \wedge g.\mathsf{tid} \neq t \wedge g.\mathsf{hole} = n \wedge \\ \mathcal{T} = \overleftarrow{\mathcal{T}} \cdot \mathtt{E}.\mathit{swap}(g.\mathsf{tid}, g.\mathsf{data}, t, n.\mathsf{data}) \end{bmatrix}_{g.\mathsf{hole},\mathcal{T}} \\
\text{FAIL}^t &\triangleq \left[\exists d.\ \mathcal{T} = \overleftarrow{\mathcal{T}} \cdot (\mathtt{E}.\{(t, \mathtt{ex}(d) \triangleright \mathsf{false}, d)\})\right]_{\mathcal{T}} \\
G^t_{\mathrm{E}} &\triangleq (\text{INIT}^t \vee \text{CLEAN}^t \vee \text{PASS}^t \vee \text{XCHG}^t \vee \text{FAIL}^t) \qquad R^t_{\mathrm{E}} \triangleq (\text{IRRELEVANT}^t_{\mathrm{E}} \vee \exists t' \neq t.\ G^{t'}_{\mathrm{ex}}) \\
J &\triangleq \forall t.\ \ g \neq \mathit{null} \wedge g.\mathsf{hole} = \mathit{null} \implies \mathrm{In}_{\mathrm{E}}(g.\mathsf{tid}) \\
A &\triangleq \mathcal{T}_{\mathrm{E}}|_{\mathtt{tid}} = T \wedge (g = \mathit{null} \vee g.\mathsf{hole} \neq \mathit{null} \vee g.\mathsf{tid} \neq \mathtt{tid}) \wedge n \mapsto \mathtt{tid}, p, \mathit{null} \\
B(k) &\triangleq (k \neq \mathit{null} \wedge k.\mathsf{tid} \neq \mathtt{tid} \wedge \mathcal{T}_{\mathrm{E}}|_{\mathtt{tid}} = T \cdot \mathtt{E}.\mathit{swap}(\mathtt{tid}, p, k.\mathsf{tid}, k.\mathsf{data}))
\end{aligned}
$$

Fig. 4. Rely/guarantee conditions and assertions used for the exchanger proof.

we take F_{E} to be the completely undefined function, which means that $\mathcal{T}_{\mathrm{E}} = \mathcal{T}|_{\mathrm{E}}$. The proof outline is shown in Figure 1. The proof uses two forms of auxiliary state. First, we instrument the code with assignments to the history variable, $\mathcal{T}$, which appears in the specification of the exchanger. We instrument the code with assignments to $\mathcal{T}$ at the successful CAS on line 29 and at the return statements on line 35. (The exact assignments we add can be read from the corresponding actions in Figure 4.) Second, we extend the `Offer` class with an auxiliary field `tid` to record the identifier of the thread that allocated the offer object. This field is used to ensure that the auxiliary assignment to $\mathcal{T}$ in the XCHG action records the correct thread identifiers.

Figure 4 defines the rely/guarantee conditions that are used in the proof. Following the trend in modern program logics [5,26], the rely/guarantee conditions are defined in terms of actions corresponding to the individual shared state updates performed. Here, actions are parametrized by the thread t performing the action. The first four actions describe the effects of the algorithm's CAS operations to the shared state, when they succeed. They modify g or $g.\mathsf{hole}$ and in the case of XCHG also the auxiliary history variable h. The FAIL action records the auxiliary assignments to h for failed exchanges, while IRR is a 'frame' action allowing other objects to append their events to h. Discarding the effects to the memory cells encapsulated by the exchanger (i.e., restricting attention to the variable h), the actions match those in the exchanger specification.

Figure 4 also defines the global invariant J saying that g cannot contain an unsatisfied offer of a thread not currently participating in the exchange, and two assertions A and B that will be used in the proof outline. We write $n \mapsto t, d, m$ as an abbreviation for $n.\mathsf{tid} = t \wedge n.\mathsf{data} = d \wedge n.\mathsf{hole} = m$. We note that J is stable both under the rely and guarantee conditions and we implicitly assume it to hold throughout execution.

We now proceed to the proof outline in Figure 1. Thanks to the encapsulated nature of concurrent objects in our programming language, we may assume that just before the start of the function $\neg\mathrm{In}_{\mathrm{E}}(\mathtt{tid})$ holds, i.e., that thread `tid` is not executing a function of E. Hence, from invariant J, we can deduce that $g = \mathit{null} \vee g.\mathsf{hole} = \mathit{null} \vee g.\mathsf{tid} \neq \mathtt{tid}$. Then after allocating the offer object, we have the assertion A. The assertion states that

the thread has not performed its operation yet, which is implied by $\mathcal{T}_{\mathtt{E}}|_{\mathtt{tid}} = T$, and that no other thread can access the newly allocated offer.

If the initialization CAS succeeds at line 15, we know that $g = n \wedge g.\mathsf{hole} = \mathit{null} \wedge \mathcal{T}_{\mathtt{E}}|_{\mathtt{tid}} = T$. This assertion, however, is not stable because another thread can come along and modify g.hole, i.e., performs the XCHG action. If this happens, then it would have made n.hole non-null and extend the history appropriately (i.e., $B(n.\mathsf{hole})$ will hold). Therefore, at line 16, the disjunction of these two assertions holds: Either an exchange has not happened, and then $n.\mathsf{hole} = \mathit{null}$, or that it was done by some other thread, and then $B(b.\mathsf{hole})$ holds.

The CAS at line 18 checks which of the above cases hold: If it succeeds, it means that waiting passively for a partner thread did not pan out. This failure, indicated by the ability to set n.hole to *fail*, is manifested in the history by extending it with the failed operation (action PASS^t). If the CAS failed than the wait did work out. Specifically, because a thread can modify the `hole` field of an offer of anther thread only when it can justify it using the XCHG action, which implies that the partner thread has also logged the successful exchange in the history variable.

Otherwise, if the initialization CAS fails, the algorithm reads g into the local variable *cur* at line 25. After this, we cannot assert that $g = \mathit{cur}$ because another thread may have modified g in the meantime. For this to happen, however, we know that *cur*.hole must be non-null; thus the disjunction $g = \mathit{cur} \vee g.\mathsf{hole} \neq \mathit{null}$ is stable. Then, if *cur* is non-null, the algorithm performs a CAS at line 29 trying to satisfy the exchange offer made by *cur*.tid. If the CAS succeeds, we know that $\mathit{cur} = g$ at the point that the CAS succeeded, and thus we can perform action XCHG and get the postcondition $B(\mathit{cur})$. Whether the CAS succeeds or not, afterwards at line 30, we know that $\mathit{cur}.\mathsf{hole} \neq \mathit{null}$, which allows us to satisfy the precondition of the CLEAN action corresponding to the final CAS operation.

6 Related Work

Neiger [18] proposed *set-linearlizability* as a means to unify specification of concurrent objects with task solutions. The main idea is to linearize concurrent operations against (a sequence of) *sets* of simultaneous operations. Neiger showed that set-linearizability is expressive enough to provide a specification for certain important tasks e.g., for Borowsky and Gafni's immediate atomic snapshot objects [2]. The notion of concurrency-aware lineraizabiity is similar to set-linearizability. Neiger, however, neither provides a formal definition of set-linearizability nor a syntactic approach to define concurrent specifications. Also, Neiger does not provide a proof technique that takes advantage of set-linearizability. In contrast, we develop a modualr proof the more general specification. In contrast, we develop all a formal proof technique for verifying concurrency-aware linearizability and employ it to produce the first compositional proof of a CA-object and of its client, namely the exchanger and the elimination stack [10]. Castaneda et al. [3] showed that set-linearizability cannot express certain tasks, e.g., write snapshot, and extended it to *interval-linearizability* which allows for arbitrary concurrent specification.

Linearizability is shown to be equivalent to observational refinement [7]. The equivalence was shown to hold even when the specification is not sequential. Thus, a direct

implication of their result is that concurrency-aware linearizability also ensures observational refinement.

The idea of elimination was introduced in [24], where it was used to construct pools and queues using trees. Example for other CA-linearizable concurrent objects can be found in [1, 11, 17, 22].

Scherer et al. present a family of *dual-data structures* [14] which support "operations that must wait for some other thread to establish a precondition". Linearizability of dual-data structures is established by explicitly specifying a "request" and "follow-up" *observable* checkpoints within the object's purview, each with its own linearization point. Dual-data structures are in fact CA-objects. We believe that using CA-histories to describe the behavior of dual data structure would help streamline their specification as it would obviate the need to specify two linearization points.

Vafeiadis [26] gives a thread modular proof for a variant of the HSY stack using RGSep [26], an extension of separation logic [19] to reason about fine-grained concurrency. His proof is not compositional as the reasoning about the elimination module is coupled with the reasoning about the stack. In particular, the elimination module is not given a context-independent specification. Dragoi et al. [6] present a technique for automatically verifying linearizability for concurrent objects are where the linearization points may be is in the body of another thread. Their technique rewrites the program to introduce combined methods whose linearization points are easy to find. They verified the elimination stack by introducing a new method `push+pop`, which simulates the elimination. As a result, their proof is inherently non compositional. In contrast, we allow for compositional proofs by (i) providing usage-context specifications for CA-object objects, (ii) allowing clients to interpret operations that seem to happen in the same point in time as an imaginary sequence of abstract operations, (iii) hiding operations on subobjects from clients of their containing object.

Sergey et al. [23] present a framework for verifying linearizability of highly concurrent data structures using time-stamped histories and subjective states, and used it to verify Hendler et al.'s flat combining algorithm. Their approach allows to hide the inter-thread interaction in the algorithm, but does not allow, at least by its current instantiations, to verify CA-linearizability. Schellhorn et al. [20] proved that backward simulation is complete for verification linearizability; it would be interesting to see if their result extends to CAL.

A novel feature of our proof technique is that it allows to relate a single concrete atomic step done by *one thread* with a sequence of abstract steps done by *multiple threads*. Our approach stands in contrast with the standard technique of using *atomicity abstraction* [4, 16, 23, 25], which allows to relate several concrete atomic actions with a single abstract step executed by *one thread*.

Acknowledgments. This research was sponsored by the EC FP7 FET project ADVENT (308830) and by Broadcom Foundation and Tel Aviv University Authentication Initiative.

References

1. Afek, Y., Hakimi, M., Morrison, A.: Fast and scalable rendezvousing. Distributed Computing 26(4), 243–269 (2013)

2. Borowsky, E., Gafni, E.: Immediate atomic snapshots and fast renaming. In: Anderson, J., Toueg, S. (eds.) PODC (1993)
3. Castaneda, A., Rajsbaum, S., Raynal, M.: Specifying concurrent problems: Beyond linearizability and up to tasks. In: DISC (2015)
4. da Rocha Pinto, P., Dinsdale-Young, T., Gardner, P.: TaDA: A logic for time and data abstraction. In: Jones, R. (ed.) ECOOP 2014. LNCS, vol. 8586, pp. 207–231. Springer (2014)
5. Dinsdale-Young, T., Dodds, M., Gardner, P., Parkinson, M.J., Vafeiadis, V.: Concurrent abstract predicates. In: ECOOP. LNCS, vol. 6183, pp. 504–528 (2010)
6. Dragoi, C., Gupta, A., Henzinger, T.A.: Automatic linearizability proofs of concurrent objects with cooperating updates. In: Sharygina, N., Veith, H. (eds.) CAV. pp. 174–190 (2013)
7. Filipovic, I., O'Hearn, P., Rinetzky, N., Yang, H.: Abstraction for concurrent objects. Theor. Comput. Sci. 411(51-52) (Dec 2010)
8. Hemed, N., Rinetzky, N.: Brief announcement: Concurrency-aware linearizability. In: Halldórsson, M.M., Dolev, S. (eds.) PODC. pp. 209–211. ACM (2014)
9. Hemed, N., Rinetzky, N., Vafeiadis, V.: Modular verification of concurrency-aware linearizability (2015), available at `http://www.cs.tau.ac.il/~nirh/disc15-ext.pdf`
10. Hendler, D., Shavit, N., Yerushalmi, L.: A scalable lock-free stack algorithm. In: SPAA (2004)
11. Hendler, D., Incze, I., Shavit, N., Tzafrir, M.: Scalable flat-combining based synchronous queues. In: Lynch, N.A., Shvartsman, A.A. (eds.) DISC. pp. 79–93 (2010)
12. Herlihy, M., Wing, J.M.: Linearizability: A correctness condition for concurrent objects. Trans. Program. Lang. Syst. 12(3), 463–492 (1990)
13. Hoare, C.A.R.: An axiomatic basis for computer programming. Commun. ACM 12(10), 576–580 (1969)
14. III, W.N.S., Scott, M.L.: Nonblocking concurrent data structures with condition synchronization. In: Guerraoui, R. (ed.) DISC 2004, LNCS, vol. 3274, pp. 174–187. Springer (2004)
15. Jones, C.B.: Specification and design of (parallel) programs. In: IFIP Congress (1983)
16. Jung, R., Swasey, D., Sieczkowski, F., Svendsen, K., Turon, A., Birkedal, L., Dreyer, D.: Iris: Monoids and invariants as an orthogonal basis for concurrent reasoning. In: POPL (2015)
17. Moir, M., Nussbaum, D., Shalev, O., Shavit, N.: Using elimination to implement scalable and lock-free fifo queues. In: SPAA. pp. 253–262. ACM (2005)
18. Neiger, G.: Set-linearizability. In: Anderson, J.H., Peleg, D., Borowsky, E. (eds.) PODC 1994. pp. 396–396. ACM (1994)
19. O'Hearn, P.W., Reynolds, J.C., Yang, H.: Local reasoning about programs that alter data structures. In: Fribourg, L. (ed.) CSL 2001. LNCS, vol. 2142, pp. 1–19. Springer (2001)
20. Schellhorn, G., Derrick, J., Wehrheim, H.: A sound and complete proof technique for linearizability of concurrent data structures. ACM Trans. Comput. Logic 15(4) (2014)
21. Scherer III, W.N., Lea, D., Scott, M.L.: A scalable elimination-based exchange channel. SCOOL (2005)
22. Scherer III, W.N., Lea, D., Scott, M.L.: Scalable synchronous queues. In: Torrellas, J., Chatterjee, S. (eds.) PPoPP 2006. pp. 147–156. ACM (2006)
23. Sergey, I., Nanevski, A., Banerjee, A.: Specifying and verifying concurrent algorithms with histories and subjectivity. In: Vitek, J. (ed.) ESOP. LNCS, vol. 9032, pp. 333–358 (2015)
24. Shavit, N., Touitou, D.: Elimination trees and the construction of pools and stacks. Theory Comput. Syst. 30(6), 645–670 (1997)
25. Svendsen, K., Birkedal, L.: Impredicative concurrent abstract predicates. In: Shao, Z. (ed.) ESOP 2014, LNCS, vol. 8410, pp. 149–168. Springer (2014)
26. Vafeiadis, V.: Modular fine-grained concurrency verification. Ph.D. thesis, University of Cambridge (2008)