A

Verifying linearizability: A comparative survey

BRIJESH DONGOL, Brunel University
JOHN DERRICK, University of Sheffield

Linearizability is a key correctness criterion for concurrent data structures, ensuring that each history of the concurrent object under consideration is consistent with respect to a history of the corresponding abstract data structure. Linearizability allows concurrent (i.e., overlapping) operation calls take effect in any order, but requires the real-time order of non-overlapping to be preserved. The sophisticated nature of concurrent objects means that linearizability is difficult to judge, and hence, over the years, numerous techniques for verifying linearizability have been developed using a variety of formal foundations such as data refinement, shape analysis, reduction, etc. However, because the underlying framework, nomenclature and terminology for each method is different, it has become difficult for practitioners to evaluate the differences between each approach, and hence, evaluate the methodology most appropriate for verifying the data structure at hand. In this paper, we compare the major of methods for verifying linearizability, describe the main contribution of each method, and compare their advantages and limitations.

Categories and Subject Descriptors: D.2.4 [**Software Engineering**]: Software/Program Verification; D.3.1 [**Programming Languages**]: Formal Definitions and Theory—*Semantics*; F.1.2 [**Computation by Abstract Devices**]: Modes of Computation—*Parallelism and concurrency*; F.3.1 [**Logics and Meanings of Programs**]: Specifying and Verifying and Reasoning about Programs—*Logics of programs*; H.2.4 [**Systems**]: Concurrency

General Terms: Algorithms, Theory, Verification

Additional Key Words and Phrases: Linearizability, concurrent objects, refinement, compositional proofs, shape analysis, reduction, abstraction, interval-based methods, mechanisation

ACM Reference Format:
Brijesh Dongol and John Derrick, 2015. Verifying linearizability: A comparative survey. *ACM Comput. Surv.* V, N, Article A (January YYYY), 44 pages.
DOI: http://dx.doi.org/10.1145/0000000.0000000

1. INTRODUCTION

Highly optimised fine-grained concurrent algorithms are increasingly being used to implement concurrent objects for modern multi/many-core applications due to the performance advantages they provide over their coarse-grained counterparts. Due to their complexity, correctness of such algorithms is notoriously difficult to judge. Formal verification has uncovered subtle bugs in published algorithms that were previously considered correct [Doherty 2003; Colvin and Groves 2005]. The main correctness criterion for concurrent algorithms is *linearizability*, which defines consistency for the history of invocation and response events generated by an execution of the algorithm at hand [Herlihy and Wing 1990]. Linearizability requires every operation call to take effect at some point between its invocation and response events. Thus, concurrent operation calls may take effect in any order, but non-overlapping operation calls must take effect

This research is supported by EPSRC Grant EP/J003727/1.

DOI: http://dx.doi.org/10.1145/0000000.0000000

in their real-time order. A (concurrent) history is linearizable iff there is some order for the effects of the operation calls that corresponds to a valid sequential history, where valid means that the sequential history can be generated by an execution of the sequential specification object. A concurrent object is linearizable iff each of its histories is linearizable.

Scalability of the proof methods for verifying linearizability remains a challenge, and hence, an immense amount of research effort has been devoted to this problem. Unfortunately, each new method is developed with respect to a specialised formal framework, making it difficult to judge the merits of the different the proof methods. Therefore, we present a comparative survey of the major techniques for verifying linearizability to examine the advantage of each method. We aim to make our comparison comprehensive, but with the scale of development in this area, it is inevitable that some published methods for verifying linearizability will be left out. Our survey does not aim to be comprehensive about fine-grained algorithms, nor about the sorts of properties that these algorithms possess; for this, [Herlihy and Shavit 2008; Moir and Shavit 2007] are already excellent resources. Instead, this survey is aimed at improving one's understanding of the fundamental challenges of linearizability verification and identifying avenues of future work. Some questions to be asked about the different methods are:

— *Locality of the proof method.* How is a proof of linearizability (a global property) decomposed so that proofs are performed in a process-local manner?
— *Compositionality of the proof method.* Does the method support compositional proofs, where interference is captured at a high level of abstraction?
— *Contribution of the framework.* Does the underlying framework contribute to simpler proofs? If so, how?
— *Algorithms verified.* Which algorithms have been verified and how complex are these algorithms?
— *Mechanisation.* Has the method been mechanised? If so, what is the level of automation?
— *Completeness.* Has completeness[1] of the proof method at hand been shown? If not, what is the verification power of each method?

Most verification techniques involve identification of a *linearization point* for each operation, which is an atomic statement of the algorithm implementing the concurrent object whose execution causes the *effect* of the operation to take place, i.e., executing a linearization point has the same effect as executing the corresponding abstract operation. It turns out that identification of linearization points is a non-trivial task. Some algorithms have simple fixed linearization points, others have external linearization points that are determined by the execution of other operations, while other yet more complex algorithms have external linearization points that potentially modify the state representation of the concurrent object. We therefore consider three case studies for comparison which are increasingly more difficult to verify — (1) an *optimistic set* with operations `add` and `remove`, both of which have fixed linearization points (2) a *lazy set* [Heller et al. 2007], which is the optimistic set together with a wait-free `contains` operation that may be linearized externally; and (3) Herlihy/Wing's array-based queue [Herlihy and Wing 1990], with future-dependent linearization points.

This paper is structured as follows. In Section 2, we present the intuition behind linearizability as well as its formal definition using Herlihy and Wing's original nomenclature. In Section 3, we present an overview of the different methods that have been

[1]A method is *complete* if whenever an implementation is linearizable with respect to an abstract specification, it can be proved linearizable using the method.

developed for verifying linearizability, which includes simulation, data refinement, auxiliary variables, shape analysis, etc. Sections 4, 5 and 6 present our case studies, where we consider algorithms for each of the different types of linearization points.

2. LINEARIZABILITY

A concurrent object allows different processes to concurrently execute its operations (by interleaving their atomic statements) so that the intervals of execution for different operation calls potentially overlap. There are numerous possible interpretations of safety for concurrent objects [Herlihy and Shavit 2008], of which, the most widely accepted condition is *linearizability* [Herlihy and Wing 1990]. We motivate linearizability using a non-blocking stack algorithm (Section 2.1) before presenting the formal definition (Section 2.2). In Section 2.3, we discuss the correspondence between linearizability and observational refinement.

2.1. Example: The Treiber stack

Fig. 1 presents a simple non-blocking stack example due to Treiber [1986], which has become a standard case study in the literature. The version we use assumes garbage collection to avoid the so-called ABA problem [Doherty 2003], where changes to shared pointers may go undetected due to the value changing from some value A to another value B then back to A. Without garbage collection, additional complexities such as version numbers for pointers must be introduced; such details are elided in this paper.

```
Init: Head = null

push(v)                              pop: lv
 H1: n := new(Node);                  P1: repeat
 H2: n.val := v;                      P2:   ss := Head;
 H3: repeat                           P3:   if ss = null then
 H4:   ss := Head;                    P4:     return empty
 H5:   n.next := ss;                  P5:   ssn := ss.next;
 H6:   until CAS(Head,ss,n)           P6:   lv := ss.val
 H7: return                           P7: until CAS(Head,ss,ssn);
                                      P8: return lv
```

Fig. 1. The Treiber stack

Treiber's stack algorithm (Fig. 1) implements the abstract stack in Fig. 2, where brackets '$\langle$' and '$\rangle$' are used to delimit sequences, '$\langle\,\rangle$' to denote the empty sequence, and '$\frown$' to denote sequence concatenation. The abstract stack consists of a shared sequence of elements S together with two operations *push* (that pushes its input $v \neq empty$ onto the top of S) and *pop* (that returns *empty* and leaves S unchanged when S is empty, and removes one element from the top of S and returns this top element otherwise).

Concurrent data structures (or more generally concurrent objects) are typically realised as part of a system library, which are instantiated in a client program, and thus the operations are assumed to be invoked by *client* processes. For reasoning purposes, one typically thinks of an object as being executed by a *most general client*, which ignores the behaviour of the clients themselves. A most general client formalises Herlihy and Wing's [Herlihy and Wing 1990] requirement that each process calls at most one operation of the object it uses at a time. For example, a most general client process of a stack [Amit et al. 2007] is given in Fig. 3, where the `?` test in the `if` is used to

model non-deterministic choice and `rand()` is assumed to return a randomly chosen non-empty element. Usage of a most general client for verification was however proposed in much earlier work [Doherty 2003].

```
Init: S = ⟨⟩

push(v)
 atomic {
    S := ⟨v⟩⌢S
 }
```

Fig. 2. An abstract stack specification

```
pop: lv
 atomic {
   if S = ⟨⟩ then
     return empty
   else
     lv := head(S);
     S := tail(S);
     return lv }
```

```
client(Stack st) {
 do {
   if (?)
     push(st, rand());
   else
     pop(st);
   } while true;
}
```

Fig. 3. Most general client process for a Stack

The implementation (Fig. 1) has fine-grained atomicity. Synchronisation is achieved using an atomic compare-and-swap (*CAS*) operation, which takes as input a (shared) variable `gv`, an expected value `lv` and a new value `nv`.

```
CAS(gv, lv, nv) ≙ atomic { if (gv = lv) then gv := nv ; return true
                           else return false }
```

In a single atomic step, the *CAS* operation compares `gv` to `lv`, potentially updates `gv` to `nv` and returns a boolean. In particular, if `gv = lv`, it updates `gv` to `nv` and returns *true* (to indicate that the update was successful), otherwise it leaves everything unchanged and returns *false*. The *CAS* instruction is natively supported by most mainstream hardware architectures. Operations that use *CAS* typically have a try-retry structure with a loop that stores (shared variable) `gv` locally in `lv`, preforms some calculations on `lv` to obtain `nv` (a new value for `gv`), then uses a *CAS* to attempt an update to `gv`. If the *CAS* fails, there must have been some interference on `gv` since it was stored locally at the start of the loop, and in this case the operation retries by re-reading `gv`.

We now explain the (concrete) program in Fig. 1, whose operations both have the try-retry structure explained above. The concrete *push* operation first creates a new node with the value to be pushed onto the stack (`H1` and `H2`). It then repeatedly sets a local variable *ss* to *Head* (`H4`) and the pointer of the new node to *ss* (`H5`) until the *CAS* succeeds (`H6`), which means *Head* (still) equalled *ss* and has atomically been set to the new node *n* (`H6`). Note that the *CAS* in *push* does not necessarily succeed: in case of a concurrent *push* or *pop* operation, *Head* might have been changed between taking the snapshot of *Head* at `H4` and execution of the *CAS* at `H6`. The concrete *pop* operation has a similar structure: it records the value of *Head* in *ss* (`P2`), and returns empty if $ss = null$ (`P4`). Otherwise, the next node is stored in *ssn* (`P5`), the return value is stored in *lv* (`P6`), and a *CAS* is executed to attempt to update *Head* (`P7`). If this *CAS* succeeds, the *pop* takes effect and the output value *lv* is returned (`P8`), otherwise, *pop* repeats its steps loading a new value of *Head*.

The linearization points of the Treiber stack are as follows. The *push* operation linearizes when the *CAS* at `H6` is successful as this is the transition that adds an element onto the top of the stack. The *pop* operation has two linearization points depending on the value returned: if the stack is empty, the linearization point is the statement labelled `P2`, when $Head = null$ is read, otherwise, the linearization point is a successful execution of the *CAS* at `P7`. Note that `P3` is not a linearization point for an empty stack

as the test only checks *local* variable *ss* — the global variable *Head* might be non-null again at this point. Notice, also, that this example illustrates the fact different statements may qualify as a linearization point depending on the values returned. In the *pop* operation, the location of the linearization point depends on whether or not the stack is empty.

A possible execution of the Treiber Stack (by a most general client) is given in Fig. 4, which depicts invocation (e.g., $push_p^I(b)$), response (e.g., $push_p^R$), and internal transitions of operations $push_p(a)$, $push_q(b)$ and $pop_r{:}\,b$, by processes p, q and r. A cross on a transition arrow is used to denote the linearization points. Although the three operations execute concurrently by interleaving their statements, the order of linearization points allows one to determine a sequential order for the operations. Importantly, this order conforms to a valid execution of the stack from Fig. 2.

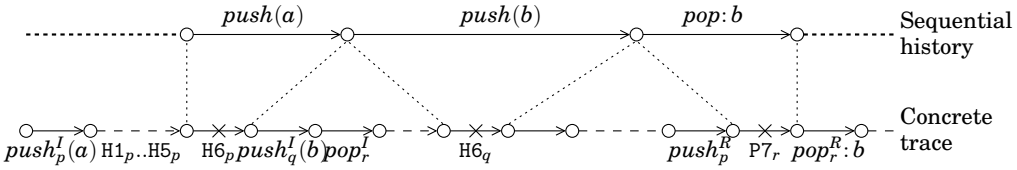

Fig. 4. Relating interleaved traces and linearizability

2.2. Formalising linearizability

Although we have motivated our discussion of linearizability in terms of the order of linearization points, and these being consistent with an abstract counterpart, we have to relate this view to what is observable in a program. In particular, what is taken to be observable are the histories, which are sequences of invocation and response events of operation calls on an object. This represents the interaction between an object and its client via the object's external interface. Thus, in Fig. 4, the internal transitions (including linearization points) are not observable.

Each observable event records the calling process (of type P), the operation that is executed (of type O), and any input/output parameters of the event (of type V). Thus, we define [Derrick et al. 2011a]:

$$Event ::= inv\langle\langle P \times O \times V \rangle\rangle \mid ret\langle\langle P \times O \times V \rangle\rangle$$

For brevity, we use notation $op_p^I(x)$ and $op_p^R{:}\,r$ for events $inv(p, op, x)$ and $ret(p, op, r)$, respectively, and use op_p^I and op_p^R to respectively denote invocation and return events with no inputs or outputs. For an event $e = (p, op, x)$, we assume the existence of projection functions $\pi_i(e)$ that returns the ith component of a tuple, e.g., $\pi_1(p, o, v) = p$.

The definition of linearizability is formalised in terms of the history of events, which is represented formally by a sequence. Namely, assuming $\mathrm{seq}(X)$ denotes sequences of type X indexed from 0 onward, a *history* is an element of $History \mathrel{\widehat=} \mathrm{seq}(Event)$, i.e., is a sequence of events.

To motivate linearizability in terms of histories, consider the following history of a concurrent stack, where execution starts with an empty stack.

$$h_1 \mathrel{\widehat=} \langle push_p^I(a), push_q^I(b), push_p^R, push_q^R \rangle$$

Processes p and q are concurrent, and hence, the operation calls may be linearized in either order, i.e., both histories below are valid linearizations.

$$hs_1 \;\widehat{=}\; \langle push_p^I(a), push_p^R, push_q^I(b), push_q^R \rangle$$
$$hs_2 \;\widehat{=}\; \langle push_q^I(b), push_q^R, push_p^I(a), push_p^R \rangle$$

Assuming execution starts with an empty stack, the abstract stack is $\langle b, a \rangle$ (with b at the top) at the end of hs_1 and $\langle a, b \rangle$ at the end of hs_2. Now suppose, history h_1 is extended with a sequential *pop* operation:

$$h_2 \widehat{=} h_1 \frown \langle pop_r^I, pop_r^R : b \rangle$$

No linearization of h_2 may swap the order of the *pop* with either of the *push* operations in h_1 because pop_r^I occurs after the return of both *push* operation calls, i.e., their executions are not concurrent. Furthermore, because elements must be inserted and removed from a stack in a last-in-first-out order, adding the *pop* that returns b restricts the valid linearizations of h_2. In particular, the only sensible choice is one in which $push(b)$ occurs after $push(a)$, i.e.,

$$hs_3 \widehat{=} hs_2 \frown \langle pop_r^I, pop_r^R : b \rangle$$

which results in an abstract stack $\langle a \rangle$ at the end of execution. Sequential history $hs_1 \frown \langle pop_r^I, pop_r^R : b \rangle$ is an invalid linearization of h_2. Now suppose h_2 is appended with two more pop operations as follows:

$$h_3 \widehat{=} h_2 \frown \langle pop_s^I, pop_t^I, pop_s^R : a, pop_t^R : a \rangle$$

History h_3 cannot be linearized by any sequential stack history — the only possible stack at the end of h_2 is $\langle a \rangle$, yet the additional events in h_3 are for two pop operations both of which are successfully able to remove a from the stack. A concurrent stack that generates h_3 would therefore be deemed incorrect. By proving the Treiber stack is linearizable, one can be assured that a history such as h_3 is never generated by the algorithm.

We now give some preliminary definitions for linearizability. For $h \in History$, let $h \mid p$ denote the subsequence of h consisting of all invocation and response events for process p. Two histories h_1, h_2 are *equivalent* if for all processes p, $h_1 \mid p = h_2 \mid p$. An index $i \in \mathrm{dom}(h)$ *matches* $j \in \mathrm{dom}(h)$ in h iff $i < j$, $h(i)$ is an invocation, $h(j)$ is a response, $\pi_1(h(i)) = \pi_1(h(j))$, $\pi_2(h(i)) = \pi_2(h(j))$ and for all $i < k < j$, $\pi_1(h(k)) \neq \pi_1(h(i))$. An invocation is *pending* in a history h iff there is no matching response to the invocation in h. We say the invocation is *complete* in h iff it is not pending in h. We let $complete(h)$ denote the maximal subsequence of history h consisting of all (completed) invocations and their matching responses in h, i.e., the history obtained by removing all pending invocations within h. For a history h, let $<_h$ be an irreflexive partial order on operations, where $op_i <_h op_j$ iff the response event of op_i occurs before the invocation event of op_j in h. A history h is *sequential* iff the first element of h is an invocation and each invocation (except possibly the last) is immediately followed by its matching response. We say that h is *well-formed* iff the subhistory $h \mid p$ is sequential. For the rest of this paper, we assume the objects in question are executed by a most general client, and hence, that the histories in question are well-formed.

Definition 2.1 (*Linearizability [Herlihy and Wing 1990]*). A history hc is *linearizable with respect to* a sequential history hs iff hc can be extended to a history hc' by adding zero or more matching responses to pending invocations such that $complete(hc')$ is equivalent to hs and $<_{hc} \subseteq <_{hs}$.

We simply say hc is linearizable if there exists a history hs such that hc is linearizable with respect to hs.

Note that Definition 2.1 allows histories to be extended with matching responses to pending invocations. This is necessary because some pending operation may have executed its linearization point, but not yet responded. For example, consider the following history, where the stack is initially empty.

$$\langle push_p^I(x), pop_q^I, pop_q^R(x)\rangle \tag{1}$$

The linearization point of $push_p^I(x)$ has clearly been executed in (1) because pop_q returns x, but (1) is incomplete because the $push_p$ is still pending. To cope with such scenarios, by the definition of linearizability, (1) may be extended with a matching response to $push_p^I(x)$, and the extended history mapped to the following sequential history: $\langle push_p^I(x), push_p^R, pop_q^I, pop_q^R(x)\rangle$.

We have defined linearizability for concurrent histories. The purpose of linearizability, however, is to define correctness of concurrent objects with respect to some abstract specification. Thus, the definition is lifted to the level of objects as follows.

Definition 2.2. A concurrent object is *linearizable* with respect to a sequential abstract specification iff for any legal history hc of the concurrent object, there exists a sequential history hs of the abstract specification such that hc is linearizable with respect to hs.

2.3. Linearizability and observational refinement

A missing link in linearizability theory is the connection between behaviours of objects and clients executing together. Namely, from a programmer's perspective, one may ask: *How are the behaviours of a client that uses a sequential object SO related to those of the client when it uses a concurrent object CO instead provided some correctness condition has been established between CO and SO?* An answer to this question was given by Filipović et al. [2010] who consider *concurrent object systems* (which are collections of concurrent objects) and establish a link between linearizability and observational refinement. Their result covers data independent clients, i.e., those that communicate only via their object systems and states that a concurrent object system *COS observationally refines* a sequential object system *AOS* iff every object in *COS* is *sequentially consistent* with respect to its corresponding object in *AOS*, where:

— *COS observationally refines AOS* iff for any client program P parameterised by an object system, the observable states[2] of $P(COS)$ is a subset of the observable states of $P(AOS)$, i.e., $P(AOS)$ does not generate any new observations in comparison to $P(COS)$, and
— *COS* is *sequentially consistent* with respect to *AOS* iff for every history h_C of *COS*, there exists a sequential history h_A such that the order of operation calls by the same process in h_C is preserved in h_A.

It is well known that linearizability implies sequential consistency, and hence, if *COS* is linearizable with respect to *AOS*, then *COS* also observationally refines *AOS* for data independent clients. In addition, Filipović et al. [2010] show equivalence between linearizability and observational refinement for clients that share data, i.e., that *COS* observationally refines *AOS* iff *COS* is linearizable with respect to *AOS*, where the definition of linearizability is suitably generalised to object systems.

[2]In their setting, the observable states consist of the variables of the clients only, i.e., none of the variables of the object system are observable.

Some authors have have presented constructive methods for developing fine-grained objects, dispensing with linearizability as a proof obligation [Turon and Wand 2011; Liang et al. 2012]. Instead, they focus on maintenance of the observable behaviour of the abstract object directly. A survey of techniques for verifying observational refinement lies outside the scope of this paper.

3. VERIFYING LINEARIZABILITY

This section discusses linearizability verification in general. Section 3.1 gives an outline of different methods for decomposing proofs, and Section 3.2 describes how linearizability verification can be characterised in terms of the linearization points. We give an overview of different methods for verifying linearizability in Sections 3.3-3.7.

3.1. Methods for proof decomposition

Capturing the correspondence between a concurrent implementation object and its sequential specification lies at the heart of linearizability. It comes as no surprise therefore that almost all methods for verifying linearizability uses some notion of *refinement* [de Roever and Engelhardt 1996] to link concrete and abstract behaviours. In this section, we classify linearizable objects based on the type of linearization point they possess, then review the different methods for proving linearizability.

Typically, the internal representation of data in a concrete object and its abstract specification differ, e.g., the Treiber stack is a linked list (Fig. 1), whereas its abstract specification is a sequence of values (Fig. 2). A formal link between their observable behaviours is given by *data refinement* [de Roever and Engelhardt 1996], which uses a *representation relation* to relate concrete and abstract state spaces. Data refinement is a system-wide (i.e., global) property and a monolithic proof of data refinement quickly becomes unmanageable. Therefore, several methods for decomposing it have been developed. The proof methods for verifying linearizability all use some combination of the methods below.

Simulation. Decomposition of data refinement into process-local proof obligations is achieved via *simulation*, which allows one to reason about each transition of the concrete object individually. Fig. 5 shows four typical simulation rules where *AInit*, *AOp* and *AFin* are abstract initialisation, operation and finalisation steps (and similarly *CInit*, *COp* and *CFin*), σ, σ' are abstract states, τ, τ' are concrete states, and *rep* is a representation relation between abstract and concrete states. Simulation proofs may be performed in a forwards or backwards manner and although the set of diagrams for forwards and backward simulation are the same, the order in which each diagram is traversed differs. In a relational setting, it turns out that neither forwards nor backwards simulation alone is complete for verifying data refinement, but the combination of the two forms a complete method [de Roever and Engelhardt 1996].[3]

Compositional frameworks. Compositional frameworks modularise reasoning about a concurrent program by capturing the behaviour of its environment abstractly. For shared-variable concurrency, a popular approach to compositionality is Jones' rely-guarantee framework [Jones 1983], where a *rely* condition states assumptions about a component's environment, and a *guarantee* condition describes the behaviour a component under the assumption that the rely condition holds. A detailed survey of different compositional verification techniques lies outside the scope of this paper; we refer the interested reader to [de Roever et al. 2001; van Staden 2015].

[3]Using a more general model of computation, e.g., predicate transformers, it is possible to develop a single complete rule for data refinement [Gardiner and Morgan 1993], but these details are elided for the purposes of this paper.

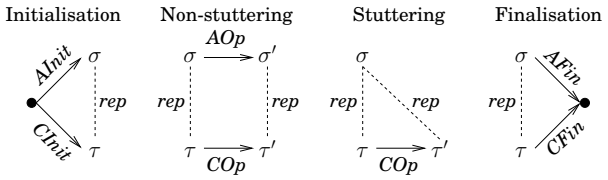

Fig. 5. Simulation diagrams

Reduction. Reduction enables one to ensure trace equivalence of the fine-grained implementation and its coarse-grained abstraction by verifying commutativity properties [Lipton 1975]. For example, in a program $S1;\ S2$ if $S2$ performs purely local modifications, $(S2_p\ ;\ T_q) = (T_q;\ S2_p)$ will hold for any statement T and processes p, q such that $p \neq q$. Therefore, $S1;\ S2$ in the program code may be treated as $atomic\{S1;\ S2\}$, which in turn enables coarse-grained atomic blocks to be constructed from finer-grained atomic statements in a manner that does not modify the global behaviour of the algorithm. After a reduction-based transformation, the remaining proof only needs to focus on verifying linearizability of the coarse-grained abstraction [Groves 2008b; 2007; Elmas et al. 2010], which is simpler than verifying the original program because fewer statements need to be considered.

Interval-based reasoning. Linearizability is a property over the intervals in which operations execute, requiring a linearization point to occur at some point between the operation's invocation and response. Some methods exploit interval logics (for example ITL [Moszkowski 2000; 1997]) to simplify reasoning. Here, a program's execution is treated as an interval predicate that defines the evolution of the system over time, as opposed to a relation that defines the individual transitions of the program.

Separation logic. Many linearizable objects are implemented using pointer-based structures such as linked lists. A well known logic for reasoning about such implementations is separation logic [Reynolds 2002; O'Hearn et al. 2001], which uses a so-called separating conjunction operator to split the memory heap into disjoint partitions, then reasons about each of these individually. Such techniques enable localised reasoning over the part of the heap that is important for the assertions at hand. Of course, pointer-based structures are not the only application of separation logic in linearizability verification, e.g., Gotsman and Yang [2013] use it to split the state spaces of an object and its clients.

The methods we discuss in this paper all use some combination of the techniques above. Prior to exploring these methods in detail, we first review the difficulties encountered when verifying linearizability.

3.2. Difficulties in verifying linearizability

One may classify different types of algorithms based on their linearization points (see Table I[4]). The type of linearization point may be distinguished as being *fixed* (i.e,. the linearization point may be predetermined), *external* (i.e,. the execution of a different operation potentially determines the linearization point) and *future-dependent* (i.e., the linearization point is determined by the future executions of the operation and in addition, these linearizations modify an the object's abstract representation). Different

[4]There are several other algorithms that have not yet been formally verified, and hence, the list of algorithms in Table I is only partial.

operations of the same object may have different types of linearization points. In fact, even within an operation, there are different types of linearization points depending on the value returned. For example, the dequeue operation of the Michael/Scott queue [Michael and Scott 1996] has both external (empty case) and fixed (non-empty case) linearization points.

Table I. Classification of algorithms

Example algorithms	Reference	Operations (linearization type)
Treiber stack	[Treiber 1986]	Push (fixed), Pop (fixed)
MS queue	[Michael and Scott 1996]	Enqueue (fixed), Dequeue (non-empty case fixed, empty case external)
Array-based queue	[Colvin and Groves 2005](1)	Enqueue (non-full case fixed, full case external), Dequeue (non-empty case fixed, empty case external)
Lock coupling list	[Herlihy and Shavit 2008]	Add (fixed), Remove (fixed)
Lazy set	[Heller et al. 2007]	Add (fixed), Remove (fixed), Contains (external)
Elimination stack	[Hendler et al. 2010]	Push (external), Pop (external)
HW queue(2)	[Herlihy and Wing 1990]	Enqueue (future), Dequeue (future)
RDCSS	[Harris et al. 2002]	Restricted double-compare single-swap (future)
CCAS	[Fraser and Harris 2007]	Conditional CAS (future)
Elimination queue	[Moir et al. 2005a]	Enqueue (future), Dequeue (future)
Snark deque	[Doherty et al. 2004a]	PushRight (future), PopRight (future), PushLeft (future), PopLeft (future)
HM lock-free set	[Michael 2002](3)	Add (true case fixed, false case fut.), Remove (true case fixed, false case fut.), Contains (external)
TSR Multiset	[Tofan et al. 2014]	Insert (fixed), Delete (future), Lookup (external)

(1) This is a corrected version of the queue by Shann et al. [2000].
(2) The dequeue operation is partial and retries as long as the queue is empty.
(3) Based on the algorithm by Harris [2001].

An example of an algorithm with fixed (or static) linearization points is the Treiber stack [Treiber 1986]. Note that these linearization points can be conditional on the global state. For example, in the `pop` operation of the Treiber stack, the statement labelled `P2` is a linearization point for the empty case if `Head = null` holds when `P2` is executed — at this point, if `Head = null` holds, one can be guaranteed that the `pop` operation will return `empty` and in addition that the corresponding abstract stack is empty. Proving correctness of such algorithms is relatively straightforward because reasoning may be performed in a forward manner. In particular, for each atomic statement of the operation, one can predetermine whether or not the statement is a linearization point and generate proof obligations accordingly. In some cases, reasoning can even be automated [Vafeiadis 2010].

An operation that has external linearization points is the `contains` operation of the lazy set by Heller et al. [2007]. The `contains` operation executing in isolation must set its own linearization points, but interference from other processes may cause it to be linearized externally. Further details of this operation are given in Section 5.1.

An example of the third class of algorithm is the queue by Herlihy and Wing [1990], where each concrete state corresponds to a set of abstract queue representations determined by the shared array and the states of all operations operating on the array. Reasoning here must be able to state properties of the form: "If in the future, the algorithm has some behaviour, then the current statement of the algorithm is a linearization point." Further complications arise when states of concrete system potentially corresponds to several possible states of the abstract data type. Hence, for each step of the concrete, one must check that each potential abstract data type is modified appropriately.

Table II presents a summary of methods for verifying linearizability, together with the algorithms that have been verified with each method and references to the papers in which the verifications are explained. Table III then presents further details of each method. The first column details whether algorithms with fixed and external linearization points have been proved, and the second details whether algorithms with future linearization points have been proved. The third column details the associated tool (if one exists), the fourth details whether the method uses a compositional approach, and the fifth details whether each method is known to be complete. The final column details whether the methods have been linked formally to Herlihy and Wing's definitions of linearizability.

3.3. Simulation-based verification

The first formal proofs of linearizability [Colvin et al. 2006a; Colvin and Groves 2005; Colvin et al. 2005; Doherty et al. 2004b; Doherty 2003] used simulation in the framework of Input/Output Automata [Lynch and Tuttle 1989]. Verification proceeds with respect to *canonical constructions* [Lynch 1996], where each operation call consists of an invocation, a single atomic transition that performs the abstract operation, and a return transition. The operations of a canonical object may be interleaved meaning its histories are concurrent, but the main transition is performed in a single atomic step. Lynch [1996] has shown that the history of every canonical construction is linearizable, and hence, any implementation that refines can be guaranteed to be linearizable.

To demonstrate this technique, consider the concrete trace from Fig. 5, recalling that the successful CAS statements at `H6` and `P7` are linearization points for the *push* and non-empty *pop* operations, respectively. One obtains the mapping between the concrete and canonical traces shown in Fig. 6. Namely, each invocation (response) transition of the concrete maps to an invocation (response) of the abstract, while a linearizing transition maps to a main transition marked in Fig. 6 by a cross. The other concrete transitions are stuttering steps (see Fig. 5), and hence, have no effect on the corresponding canonical state.

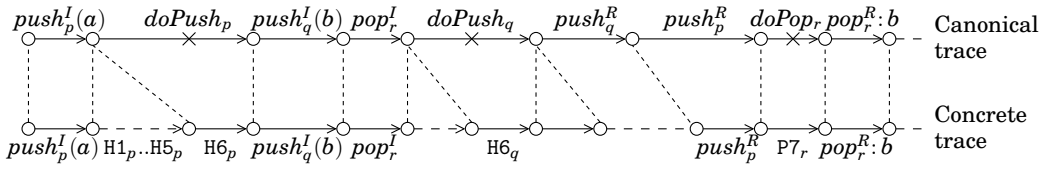

Fig. 6. Groves et al.'s simulation proofs for linearizability

Table II. Methods for verifying linearizability

Method	Algorithms verified	Reference
Canonical abstraction(1)	Treiber stack	[Groves 2009]
	MS queue	[Doherty et al. 2004b](2)
	Array-based queue	[Colvin and Groves 2005]
	Lazy set	[Colvin et al. 2006a]
	Elimination stack	[Groves and Colvin 2007]
	Snark double-ended queue	[Doherty et al. 2004a]
Sequential abstraction	Treiber stack, Lock-coupling set	[Derrick et al. 2011a]
	Lazy set	[Derrick et al. 2011b]
	HW queue	[Schellhorn et al. 2014]
Rely-guarantee and separation logic	RDCSS, Lock-coupling set, Optimistic set, Lazy set	[Vafeiadis 2007] and [Liang and Feng 2013a]
	MCAS	
	CCAS, Elimination stack, Two-lock queue, MS queue(2), HM lock-free set(4)	[Vafeiadis 2007] [Liang and Feng 2013a]
Reduction	Treiber stack	[Elmas et al. 2010; Groves 2009]
	MS queue	[Groves 2008b; Elmas et al. 2010]
	Elimination stack	[Groves and Colvin 2009]
	Simplified multiset	[Elmas et al. 2010]
Rely-guarantee and interval temporal logic	Treiber stack, MS queue(2)	[Bäumler et al. 2011]
	Treiber stack with hazard pointers	[Tofan et al. 2011]
	TSR multiset	[Tofan et al. 2014]
Shape analysis	Treiber stack, MS queue(2)	[Amit et al. 2007]
	Numerous algorithms from [Herlihy and Shavit 2008]	[Vafeiadis 2010]
Construction-based	Treiber stack	[Jonsson 2012]
	MS queue	[Abrial and Cansell 2005] and [Groves and Colvin 2009]
	Elimination stack	[Groves and Colvin 2007]
	Optimistic set	[Vechev and Yahav 2008]
Hindsight lemma	Optimistic set(3), Lazy set(3)	[O'Hearn et al. 2010a; 2010b]
Interval abstraction	Lazy set	[Dongol and Derrick 2013]
Aspect-oriented proofs	HW queue	[Henzinger et al. 2013b]

(1) This is the only method known to have found two bugs in existing algorithms [Doherty 2003; Colvin and Groves 2005].
(2) Including a variation by Doherty et al. [2004b].
(3) The use of atomicity brackets prohibits behaviours that are permitted by the fine-grained algorithm.
(4) Set algorithm by Michael [2002], which is based on the algorithm by Harris [2001].

Although Groves et al. present a sound method for proving linearizability, a fundamental question about the link between concurrent and sequential programs remains. *Can linearizability be formulated as an instance of data refinement between a concurrent implementation and a sequential abstract program?* This is answered by Derrick et al. [2011a], who present a simulation-based method linking the concurrent object in question with its sequential (as opposed to canonical) abstraction. This is achieved by including an auxiliary history variable in the states of both the concrete and abstract objects so that linearizability is established as part of the refinement. In addition, a number of process-local proof obligations that dispense with histories are generated, whose satisfaction implies linearizability. Instead of proving refinement in a layered

Table III. Comparison of verification methods

Method	Fixed & External	Future	Tool	Compositional?	Complete?	Linked to HW
Canonical abstraction	✓	✓	PVS		(1)	(4)
Sequential abstraction	✓	✓	KIV		(2)	✓
RG+SL	✓	✓		✓	(3)	(5)
Reduction	✓		QED			
RGITL	✓	✓	KIV	✓	(2)	(6)
Shape analysis	✓		CAVE			
Construction-based	✓					
Hindsight lemma	✓					
Interval abstraction	✓			✓		
Aspect-oriented proofs		✓	CAVE			(7)

(1) Forwards and backwards simulation is complete for showing refinement of input/output automata [Lynch and Vaandrager 1995].
(2) Backward simulation for history-enhanced data types shown to be complete for linearizability [Schellhorn et al. 2012; 2014].
(3) Completeness could potentially be proved by linking these methods the results of Abadi and Lamport [1991], however, this link has thus far not been made.
(4) Using results of Lynch [1996].
(5) Using results in [Liang and Feng 2013b; 2013a].
(6) Using an alternative characterisation of linearizability based on *possibilities* [Herlihy and Wing 1990].
(7) Applies purely blocking implementations only.

manner (as done by Groves et al.), Derrick et al.'s proofs aim to capture the relationships between the abstract and concrete systems within the refinement relation itself.

For a concrete example, once again consider the stack trace from Fig. 5. Using the methods of Derrick et al. [2011a], one would obtain a refinement shown in Fig. 7, where the concrete transitions that update the history are indicated with a bold arrow. Assume hc and ha are the concrete and abstract history variables, both of which are sequences of events. Each concrete invoke or return transition appends the corresponding event to the end of hc, e.g., transition $push_p^I(a)$ updates hc to $hc \frown \langle push_p^I(a) \rangle$. Every abstract transition updates the ha with matching invocation and response pairs, e.g., $APush_p$ updates the ha to $ha \frown \langle push_p^I(a), push_p^R \rangle$. Therefore, the concrete history hc may be concurrent, whereas the abstract history ha is sequential. This enables the proof of linearizability to be built into the refinement relation, as opposed to relying on a canonical abstraction that generates linearizable histories.

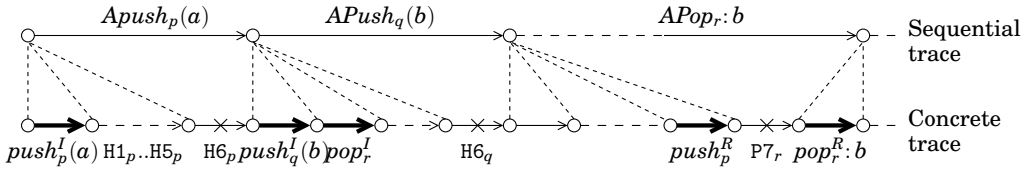

Fig. 7. Derrick et al.'s refinement proofs for linearizability

3.4. Augmented states

Instead of defining concrete and abstract objects as separate systems and using a representation relation to link their behaviours (as done in Section 3.3), one may embed the abstract system directly within the concrete system as an auxiliary extension [Vafeiadis 2007] and prove linearizability by reasoning about this combined system. For example, in a proof of the Treiber stack, one would introduce the abstract sequence S as an auxiliary variable to the program in Section 1. At each linearization point of the Treiber stack, a corresponding operation is performed on S, e.g., the successful `CAS` transition at `H6` is augmented so that S is updated to $\langle v \rangle \frown S$ [Vafeiadis 2007]. This has the advantage of flattening the state space into a single layer meaning proofs of linearizability follow from invariants on the combined state. Vafeiadis [2007] further simplifies proofs by using a framework that combines *separation logic* [O'Hearn et al. 2001] (to reason about pointers) and *rely-guarantee* [Jones 1983] (to support compositionality). It is worth noting, however, the underlying theory using this method relies on refinement [Liang and Feng 2013b]. Namely, the augmentation of each concrete state must be an appropriate abstraction of the concrete object in question.

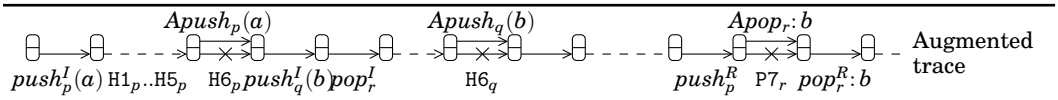

Fig. 8. Vafeiadis et al.'s augmented state based proofs

To visualise this approach, again consider the example trace from Fig. 4, where embedding the abstract state as an auxiliary variable produces the augmented trace in Fig. 8. For algorithms with fixed linearization points (which can be verified using forward simulation), reasoning about invariants over the flattened state space is simpler than simulation proofs. (This is also observed in the forward simulation proof of Colvin et al. [2006a], where auxiliary variables that encode the abstract state are introduced at the concrete level.) However, invariant-based proofs only allow reasoning about a single state at a time, and hence are less flexible than refinement relations, which relate a concrete state to potentially many abstract states. Vafeiadis [2007] addresses these shortcomings by using more sophisticated auxiliary statements that are able to linearize both the currently executing operation as well as other executing processes. In addition, *prophecy variables* [Abadi and Lamport 1991] are used to reason about operations whose linearization points depend on future behaviour. Recently, Liang and Feng [2013b] have consolidated these ideas augmentations by allowing auxiliary statements `linself`, (which performs the same function as the augmentations of Vafeiadis by linearizing the currently executing process [Vafeiadis et al. 2006]) and `lin(p)`, (which performs the linearization of process p different from *self* that may be executing a different operation). Liang and Feng (unlike Vafeiadis) allow augmentations that use `try` and `commit` pairs, where the `try` is used to guess potential linearization points, and the `commit` used to pick from the linearization points that have been guessed thus far.

Augmented state spaces also form the basis for *shape analysis* [Jones and Muchnick 1982], which is a static analysis technique for verifying properties of objects with dynamically allocated memory. One of the first shape-analysis-based linearizability proofs is that of Amit et al. [2007], who consider implementations using singly linked lists and fixed linearization points. The following paraphrases Amit et al. [2007, pg 480], by clarifying their nomenclature with the terminology used in this paper.

> The proof method uses a correlating semantics, which simultaneously manipulates two memory states: a so-called *candidate state* [i.e., concrete state] and the *reference state* [i.e., abstract state]. The candidate state is manipulated according to an interleaved execution and whenever a process reaches a linearization point in a given procedure, the correlating semantics invokes the same procedure with the same arguments on the reference state. The interleaved execution is not allowed to proceed until the execution over the reference state terminates. The reference response [i.e., return value] is saved, and compared to the response of the corresponding candidate operation when it terminates. Thus linearizability of an interleaved execution is verified by constructing a (serial) witness execution for every interleaved execution.

These methods are extended by Vafeiadis [2009], where a distinction is made between shape abstraction (describing the structure of a concurrent object) and value abstraction (describing the values contained within the object). The method is used to verify several algorithms, including the complex RDCSS algorithm with future linearization points.

Although the behaviours of concurrent objects are complex, the algorithms that implement them are often short, consisting of only a few lines of code. This makes it feasible to perform a brute-force search for their linearization points. To this end, Vafeiadis [2010] presents a fully automated method that infers the required abstraction mappings based on the given program and abstract specification of the objects. The method is, thus far, only able to handle so-called *logically pure* operations. An example of a logically impure operation is the `remove` operation of the optimistic set (Section 4.1), which uses a special "marked bit" to denote nodes that have been logically removed from the set.

3.5. Interval-based methods

Interval-based methods aim to treat programs as executing over an interval of time, as opposed to relations between pre and post states. Schellhorn et al. combine rely-guarantee reasoning with interval temporal logic [Moszkowski 2000], which enables one to reason over the interval of time in which a program executes, as opposed to single state transition [Schellhorn et al. 2011]. The proofs are carried out using the KIV theorem prover [Drexler et al. 1993], which is combined with symbolic execution [Burstall 1974; Bäumler et al. 2010] to enable guarantee conditions to be checked. This involves inductively stepping through the program statements within KIV itself, simplifying verification. These methods have been applied to verify the Treiber stack and the Michael/Scott queue [Bäumler et al. 2011].

Dongol and Derrick [2013] verify behaviour refinement between a coarse-grained abstraction and fine-grained implementation. Unlike all other methods, these proofs do not rely on identification of linearization points in the concrete code. The method has been applied to the lazy set algorithm [Heller et al. 2007], including the `contains` operation with external linearization points.

3.6. Problem-specific techniques.

Researchers have also developed problem-specific methods, sacrificing generality in favour of simpler linearizability proofs for a specific subset of concurrent objects. One such method for non-blocking algorithms is the Hindsight Lemma [O'Hearn et al. 2010a], which applies to linked list implementations of concurrent sets (e.g., the lazy set) and characterises conditions under which a node is guaranteed to have been in or out of a set. The original paper [O'Hearn et al. 2010a] only considers a simple opti-

mistic set. The extended technical report [O'Hearn et al. 2010b] presents a proof of the Heller et al.'s lazy set. Unfortunately, the locks within the add and remove operations are modelled using atomicity brackets, which has the unwanted side effect of disallowing concurrent reads of the locked nodes. Thus, the algorithm verified by O'Hearn et al. [2010b] differs operationally from the Heller et al. lazy set [Heller et al. 2007]. Overall, the ideas behind problem-specific simplifications such as the Hindsight Lemma are interesting, but the logic used and the objects considered are highly specialised.

Some objects like queues and stacks can be uniquely identified by their *aspects*, which are properties that uniquely characterise the object in question. This is exploited by Henzinger et al. [2013b], who present an aspect-oriented proof of the Herlihy/Wing queue. Further details of this particular method are provided in Section 6.2.

Automation has been achieved for algorithms with helping mechanisms and external linearization points such as the elimination stack [Dragoi et al. 2013]. These techniques require the algorithms to satisfy so-called R-linearizability [Pacull and Sandoz 1993], a stronger condition than linearizability, hence, verification of algorithms with linearization points based on future behaviour are excluded.

3.7. Construction-based proofs

Several researchers have also proposed the development of linearizable algorithms via incremental refinement, starting with an abstract specification. Due to the transitivity of refinement, and because the operations of the initial program are atomic (and trivially linearizable), linearizability of the final program is also guaranteed. An advantage of this approach is the ability to *design* an implementation algorithm, leaving open the possibility of developing variations of the desired algorithm.

The first constructive approach to linearizability is by Abrial and Cansell [2005], who use the Event-B framework [Abrial 2010] and the associated proof tool. However, the final algorithm they obtain requires counters on the nodes (as opposed to pointers [Michael and Scott 1996]), thus it is not clear whether such a scheme really is implementable. Groves [2008a] presents a derivation of the Michael/Scott queue using reduction to justify each refinement step [Lipton 1975]. This is extended by Groves and Colvin [2009], who derive a more complicated stack by Hendler et al. [2010] that uses an additional "backoff array" in the presence of high contention for the shared central stack. Their derivation methods allow data refinement (without changing atomicity), operation refinement (where atomicity is modified, but state spaces remain the same) and refactoring (where the structure of the program is modified without changing its logical meaning) [Groves and Colvin 2009; 2007]. These proofs are not mechanised, but there is potential to perform mechanisation using proof tools such as QED [Elmas 2010].

Gao et al. [2009; 2007; 2005] present a number of derivations of non-blocking algorithms and develop a number of special-purpose reduction theorems for derivation [Gao and Hesselink 2007]. However, these derivations aim to preserve *lock-freedom* (a progress property) [Massalin and Pu 1992], as opposed to linearizability.

Vechev et al. [2008; 2009] present a tool-assisted derivation method based on bounded model checking. Starting with a sequential linked-list set, they derive several variations of set algorithms implemented using DCAS (double compare-and-swap) and CAS instructions, as well as variations that use marking schemes. Although their methods allow relatively large state spaces to be searched, these state spaces are bounded in size, and hence, only finite executions by a fixed number of processes are checked, i.e., linearizability of the final algorithms derived cannot be guaranteed.

More recently, Jonsson [2012] has presented a derivation of the Treiber stack and Michael/Scott queue in a refinement calculus framework [Morgan and Vickers 1992]. Jonsson defines linearizability as

Table IV. Overview of methods for verifying set algorithms

Reference	Lin. point identification	Additional notes
[Vafeiadis et al. 2006]	Manual	Operation `contains` not verified
[Colvin et al. 2006a]	Manual	Allows model checking
[Vafeiadis 2007]	Manual	Auxiliary code can linearize other operations
[Vafeiadis 2010]	Automatic	Full automation via shape analysis, but the lazy set [Heller et al. 2007] is not yet verified in the method.
[O'Hearn et al. 2010a]	N/A	Uses Hindsight Lemma to generate proof obligations, and hence, only applicable to list-based set implementations
[Elmas et al. 2009]	N/A	Linearizability proofs are performed for coarse-grained abstractions
[Derrick et al. 2011b]	Manual	Data refinement proofs
[Liang and Feng 2013a]	Manual	Separation logic encoding
[Dongol and Derrick 2013]	N/A	Interval-based reasoning; linearizability is proved for coarse-grained abstractions

> A program P is linearizable if and only if $atomic\{P\}$ is refined by P. [Jonsson 2012, Definition 3.1]

Reduction-style commutativity checks are used to justify splitting the atomicity at each stage. With this interpretation of linearizability, Jonsson is able able to start by treating the entire concrete operation as a single atomic transition, then incrementally split its atomicity into finer-grained statements.

4. CASE STUDY 1: AN OPTIMISTIC SET ALGORITHM

Set algorithms have become standard case studies for showing applicability of a theory for verifying linearizability. Of particular interest is the lazy set by Heller et al. [2007], which is a simple algorithm with `add` and `remove` operations that have fixed linearization points and a `contains` operation that is potentially linearized by the execution of other operations. We first present a verification of a simplified version that consists of `add` and `remove` operations only. An overview of the different approaches to verifying set algorithms is given in Table IV. Further details of each method are provided in the sections that follow. The formalisation in this section aims to highlight the main ideas behind each method. We refer readers interested in reproducing each proof to the original papers.

4.1. An optimistic set

In this section, we present a simplified version of Heller et al.'s concurrent set algorithm [Heller et al. 2007] (see Fig. 9) operating on a shared linked list, that is sorted in strictly ascending values order. Locks are used to control concurrent access to list nodes. The algorithm consists of operations `add` and `remove` that use auxiliary operation `locate` to optimistically determine the position of the node to be inserted/deleted from the linked list.

Each node of the list consists of fields $val, next, mark$, and $lock$, where val stores the value of the node, $next$ is a pointer to the next node in the list, $mark$ denotes the

```
add(x):
 A1: n1, n3 := locate(x);
 A2: if n3.val != x then
 A3:   n2 := new Node(x);
 A4:   n2.next := n3;
 A5:   n1.next := n2;
 A6:   res := true
     else
 A7:   res := false
 A8: n1.unlock();
 A9: n3.unlock();
A10: return res
```

```
remove(x):
 R1: n1, n2 := locate(x);
 R2: if n2.val = x then
 R3:   n2.mark := true;
 R4:   n3 := n2.next;
 R5:   n1.next := n3;
 R6:   res := true
     else
 R7:   res := false;
 R8: n1.unlock();
 R9: n2.unlock();
R10: return res
```

```
locate(x):
     while true do
 L1:   pred := Head;
 L2:   curr := pred.next;
 L3:   while curr.val < x do
 L4:     pred := curr;
 L5:     curr := pred.next
 L6:   pred.lock();
 L7:   curr.lock();
 L8:   if !pred.mark
         and !curr.mark
         and pred.next = curr
 L9:   then return pred, curr
       else
L10:    pred.unlock();
L11:    curr.unlock()
```

Fig. 9. Optimistic set algorithm operations

marked bit[5] and *lock* stores the identifier of the process that currently holds the lock to the node (if any). The *lock* field of each node only prevents modification to the node; it is possible for processes executing `locate` and `contains` to read values of locked nodes when they traverse the list. Two dummy nodes with values $-\infty$ and ∞ are used at the start (`Head`) and end (`Tail`) of the list, and all values v inserted into the set are assumed to satisfy $-\infty < v < \infty$.

Operation `locate(x)` is used to obtain pointers to two nodes `pred` (the predecessor node) and `curr` (the current node). A call to `locate(x)` operation traverses the list ignoring locks, acquires locks once a node with value greater than or equal to `x` is reached, then *validates* the locked nodes. If the validation fails, the locks are released and the search for `x` is restarted. When `locate(x)` returns, both `pred` and `curr` are locked by the calling process, the value of `pred` is always less than `x`, and the value of `curr` may either be greater than `x` (if `x` is not in the list) or equal to `x` (if `x` is in the list).

Operation `add(x)` calls `locate(x)`, then if `x` is not already in the list (i.e., value of the current node `n3` is strictly greater than `x`), a new node `n2` with value field `x` is inserted into the list between `n1` and `n3` and *true* is returned. If `x` is already in the list, `add(x)` does nothing and returns *false*. Operation `remove(x)` also starts by calling `locate(x)`, then if `x` is in the list the current node `n2` is removed and *true* is returned to indicate that `x` was found and removed. If `x` is not in the list, the `remove` operation does nothing and returns *false*. Note that operation `remove(x)` distinguishes between a logical removal, which sets the marked field of `n2` (the node corresponding to `x`), and a physical removal, which updates the `next` field of `n1` so that `n2` is no longer reachable.

As a concrete example, consider the linked list in Fig. 10 (a), which represents the set $\{3, 18, 77\}$, and an execution of `add(42)` by process p without interference. Execution starts by calling `locate(42)` and once this returns, $n1_p$ and $n2_p$ are set as shown in Fig. 10 (b). Having found and locked the correct location for the insertion, process p tests to see that the value is not already in the set (line `A2`), then creates a new unmarked node $n3_p$ with value 42 and next pointer $n3_p$ (see Fig. 10 (c)). Then by executing `A4`, the executing process sets the next pointer of $n1_p$ to $n2_p$, linearizing a successful `add` operation (see Fig. 10 (d)). Thus, provided no `remove(42)` operations are executed, any other `add(42)` operation that is started after `A4` has been executed will return *false*.

[5]The *mark* bit is not strictly necessary to implement the optimistic set (e.g., [Vafeiadis 2007]), however, we use it here to simplify the lead up to the lazy set in Section 5.

After the linearization, process p releases the locks on $n1_p$ and $n3_p$ and returns *true* to indicate the operation was successful.

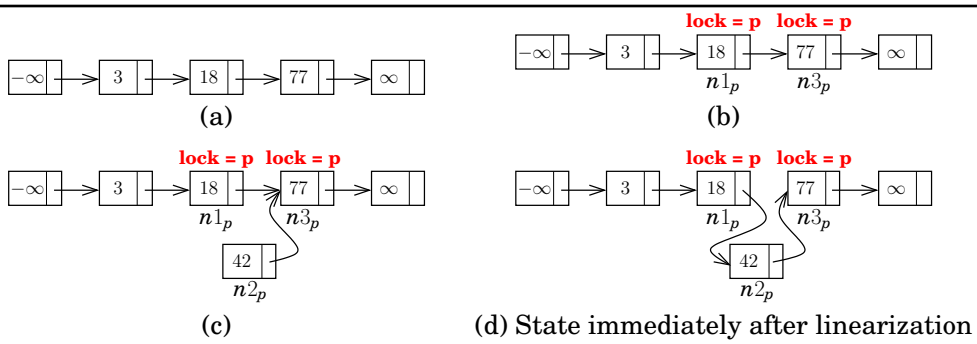

Fig. 10. Execution of `add(42)` by process p

Now consider the execution of `remove(18)` by process p on the set $\{3, 18, 77\}$ depicted by the linked list in Fig. 11 (a), where the process executes without interference. Like `add`, operation `remove(18)` operation first calls `locate(18)`, which returns the state depicted in Fig. 11 (b). At `R2`, a check is made that the element to be removed (given by node $n2_p$) is actually in the set. Then, the node $n2_p$ is removed logically by setting its marked value to *true* (line `R3`), which is the linearization point of `remove` (see Fig. 11 (c)). After execution of the linearization point, operation `remove` sets $n3_p$ to be the next pointer of the removed node (line `R4`), and then node $n2_p$ is physically removed by setting the next pointer of `n1` to $n3_p$ (see Fig. 11 (d)). Then, the held locks are released and *true* is returned to indicate that the `remove` operation has succeeded. Note that although 18 has been logically removed from the set in Fig. 11 (c), no other process is able to insert 18 to the set until the marked node has also been physically removed (as depicted in Fig. 11 (d)), and the lock on $n1_p$ has been released.

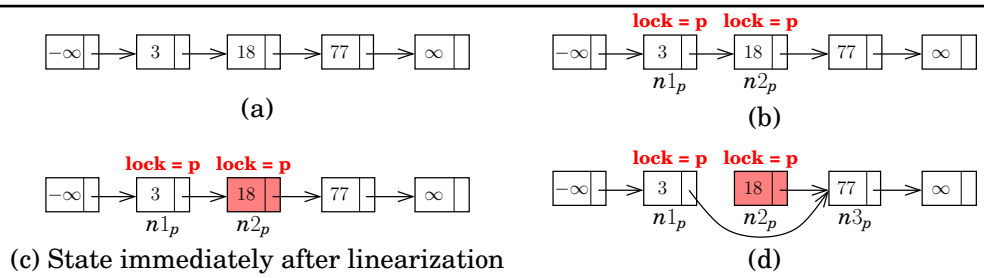

Fig. 11. Execution of `remove(18)` by process p

Verifying `add` *and* `remove` *operations.* Verifying correctness of `add` and `remove`, which have fixed linearization points, is relatively straightforward because the globally visible effect of both operations may be determined without having to refer to the *future states* of the linked list. The refinement-based methods (Section 3.3) verify correctness using forward simulation and the state augmentation methods (Section 3.4) modify the abstract state directly.

We present outlines of the proofs using the simulation-based methods of Colvin et al. [2006a] (Section 4.2), refinement-based method of Derrick et al. [2011b] (Section 4.3) and auxiliary variable method of Vafeiadis [2007] (Section 4.4). To unify the presentation, we translate the PVS formulae from [Colvin et al. 2006b] and the Vafeiadis' RGSep notation [Vafeiadis and Parkinson 2007; Vafeiadis 2007] into Z [Bowen 1996], which is the notation used by Derrick et al. Inevitably, this causes some of the benefits of a proof method to be lost; we discuss the effect of the translation and the benefits provided by the original framework where necessary.

Full details on modelling concurrent algorithms in Z are given by Derrick et al. [2011a]. To reason about linked lists, memory must be explicitly modelled, and hence, the concrete state *CState* is defined as follows, where *Label* and *Node* are assumed to be the types of a program counter label and node, respectively. Each atomic program statement is represented by a Z schema. The schema for the statements in Fig. 9 labelled `A5` and `A7` executed by process p are modelled by $Add5_p$ and $Add7_p$, respectively. Notation $\Delta CState$ imports both unprimed and primed version of the variables of *CState* into the specification enabling one to identify specifications that modify *CState*; unprimed and primed variables are evaluated in the current and next states, respectively. Using the Object-Z [Smith 1999] convention, we assume that variables $v' = v$ for every variable v unless $v' = k$ is explicitly defined for some value k.

$$
\begin{array}{|l}
\hline
CState \\
\hline
pred, curr : P \to Node \\
n1, n2, n3 : P \to Node \\
pc : P \to Label \\
lock : Node \to \mathbb{P}\, P \\
next : Node \to Node \\
mark : Node \to \mathbb{B} \\
res : P \to V \\
\hline
\end{array}
\qquad
\begin{array}{|l}
\hline
Add5_p \\
\hline
\Delta CState \\
\hline
pc(p) = A5 \\
next'(n1(p)) = n2(p) \\
\hline
\end{array}
\qquad
\begin{array}{|l}
\hline
Add7_p \\
\hline
\Delta CState \\
\hline
pc(p) = A7 \\
res'(p) = false \\
\hline
\end{array}
$$

4.2. Method 1: Proofs against canonical specifications

This section reviews Groves et al.'s simulation methods against canonical specifications [Doherty et al. 2004b]. Here, one is required to perform the following steps.

(1) Identify and fix the linearization points of each concrete operation.
(2) Define a canonical abstraction and a representation relation that describes the link between the canonical and concrete representations.
(3) Prove simulation between the concrete program (which is the program in Fig. 9 formalised in Z) and canonical abstraction, where the concrete initialisation and responses are matched with abstract initialisation and response operations, respectively. The linearization points must be matched with main canonical operations. Simulation may be performed in a forwards or backwards manner, and in some cases, both are required (whereby several intermediate layers of abstraction may be introduced). The proof may require introduction of additional invariants at the concrete level to specify additional properties of the data structure in question.

The linearization points have been described in Section 4.1. To model the canonical specification, first the abstract state *AState* must be defined.

$$
\begin{array}{|l}
\hline
AState \\
\hline
S : \mathbb{P}\, V \\
pc : P \to Label \\
v : P \to V \\
res : P \to \mathbb{B} \\
\hline
\end{array}
$$

The canonical operations corresponding to the `add` operation are given by the following Z schema, where variables decorated with ? and ! denote inputs and outputs, respectively.

$$
\begin{array}{|l}
\hline
AddInv_p \\
\Delta AState \\
x? \in V \\
\hline
pc(p) = idle \\
pc'(p) = addi \\
v'(p) = x? \\
\hline
\end{array}
\qquad
\begin{array}{|l}
\hline
AddOK_p \\
\Delta AState \\
\hline
pc(p) = addi \\
v(p) \in S \\
S' = S \cup \{v(p)\} \\
res'(p) = true \\
pc'(p) = addo \\
\hline
\end{array}
\qquad
\begin{array}{|l}
\hline
AddFail_p \\
\Delta AState \\
\hline
pc(p) = addi \\
v(p) \notin S \\
res'(p) = false \\
pc'(p) = addo \\
\hline
\end{array}
\qquad
\begin{array}{|l}
\hline
AddRes_p \\
\Delta AState \\
r! \in \mathbb{B} \\
\hline
pc(p) = addo \\
r! = res(p) \\
pc'(p) = idle \\
\hline
\end{array}
$$

Similar schema are generated for the canonical form of the `remove` operation. Following Lynch [1996], any history generated by such canonical specifications are linearizable, and therefore, any refinement of the canonical specification must also be linearizable.

As highlighted in Section 5, the forward simulation must consider four different simulation diagrams: initialisation, stuttering and non-stuttering transitions, and finalisation. For the non-stuttering transitions, which are the most interesting of these, the forward simulation proof rule states the following, where AOp_p is the abstract operation corresponding to the COp_p in process p, rep is a relation from the abstract to the concrete state space, and '$\fatsemi$' denotes relational composition, i.e., for relations $r_1 \in V_X \leftrightarrow V_Y$ and $r_2 \in V_Y \leftrightarrow V_Z$, we define $r_1 \fatsemi r_2 = \{(x,z) \mid x \in V_X \land z \in V_Z \land \exists y : V_Y \bullet (x,y) \in r_1 \land (y,z) \in r_2\}$.

$$\forall p : P \bullet rep \fatsemi COp_p \subseteq AOp_p \fatsemi rep \tag{2}$$

Thus, for any abstract state σ and concrete state τ linked by the representation relation rep, if the concrete statement COp_p is able to transition from τ to τ', then there must exist an abstract state σ' such that AOp_p can transition from σ to σ' and σ' is related to τ' via rep.

Colvin et al. [2006a] set up a framework that enables model checking of possible invariants prior to its formal verification in a theorem prover. To this end, auxiliary variables that reflect the abstract space are introduced at the concrete level together with invariants over these auxiliary variables that correspond to the simulation relation. For the lazy set, one such variable is aux_S, which stores the set of elements currently in the set. The set aux_S is updated whenever a node is inserted into the list, or is marked for deletion. To verify that aux_S does indeed represent the abstract set, one must prove that the following holds:

$$cs(aux\_S) = \{k \in V \mid InList(cs, k)\}$$

where cs is a reachable concrete state and $InList$ is a function that determines whether or not the value k is in the list (i.e., an unmarked node with value k is reachable from the head). Using aux_S, the main invariants that Colvin et al. prove are:[6]

$$\forall p : P \bullet pc(p) \in \{A5, R7\} \Rightarrow v(p) \notin aux\_S \tag{3}$$

$$\forall p : P \bullet pc(p) \in \{A7, R3\} \Rightarrow v(p) \in aux\_S \tag{4}$$

By (3), for any process p, prior to execution of execution of `A5` (a successful `add`) and `R7` (a failed `remove`), the element being added and removed, respectively must not be in the set. Condition (4) is similar. The representation relation between an abstract state

[6] In [Colvin et al. 2006b] $A5$ and $A7$ are labelled $add6$ and $add8$, respectively.

as and concrete state cs is defined as follows, where *step_rel* is a relation between the program counters of as and cs.

$$rep(as, cs) \mathrel{\widehat=} as(S) = cs(aux\_S) \land step\_rel(as, cs)$$

Proofs of these conditions require a number of additional invariants to be established, e.g., stating that the list is sorted. However, it is worth noting that a substantial number of these invariants are introduced to prove the full lazy set. These proofs are carried out entirely within PVS [Owre et al. 1996].

4.3. Method 2: Proofs against sequential specifications

Derrick et al.'s method considers proofs directly against a sequential specification (with atomic operations), unlike the previous method that verifies refinement against a canonical specification with additional invoke/return transitions. Verification using this method consists of the following steps.

(1) Identify and fix the linearization points of each operation.
(2) Decompose the proof into process-local proof obligations using a *status* function.
(3) Prove, using simulation, that each concrete step is a refinement of some abstract step.
(4) Show that other processes running in parallel maintain the refinement relation. To this end, encode the *interference freedom* and *disjointness* proof obligations within the invariants.
(5) Finally, prove that the initialisation establishes the refinement relation.

The abstract state and operations `add` and `remove` are modelled as follows:

$$\begin{aligned} AState &\mathrel{\widehat=} [S : \mathbb{P}\, V] \\ Add_p &\mathrel{\widehat=} [\Delta AState, x? : V, r! : \mathbb{B} \mid S' = S \cup \{x?\} \land r! = (S' \neq S)] \\ Remove_p &\mathrel{\widehat=} [\Delta AState, x? : V, r! : \mathbb{B} \mid S' = S \setminus \{x?\} \land r! = (S' \neq S)] \end{aligned}$$

The proofs rely on *history-enhanced objects*, which introduce the sequential and concrete histories as auxiliary variables. Executing operations append events to a history, e.g., an invocation op with input x executed by process p, appends $inv(p, op, x)$ to the history. Thus, if h is the concrete history variable, the invocation and return schema of the `add` operation are extended as follows:

$$\begin{aligned} AddInvH_p &\mathrel{\widehat=} AddInv_p \land [h, h' : \mathrm{seq}(Event) \mid h' = h \frown \langle inv(p, add, x?)\rangle] \\ AddRetH_p &\mathrel{\widehat=} AddRet_p \land [h, h' : \mathrm{seq}(Event) \mid h' = h \frown \langle ret(p, add, r!)\rangle] \end{aligned}$$

The abstract data types execute the operations atomically, and hence, their invocation and return occur as part of a single transition. Given that hs is the auxiliary sequential history variable, the following schema formalise the history-enhanced abstract add and remove operations:

$$\begin{aligned} AddH_p &\mathrel{\widehat=} Add_p \land [hs, hs' : \mathrm{seq}(Event) \mid hs' = hs \frown \langle inv(p, add, x?), ret(p, add, r!)\rangle] \\ RemH_p &\mathrel{\widehat=} Remove_p \land [hs, hs' : \mathrm{seq}(Event) \mid hs' = hs \frown \langle inv(p, rem, x?), ret(p, rem, r!)\rangle] \end{aligned}$$

Therefore, the abstract history is sequential, whereas the concrete is concurrent. Refinement between the abstract and concrete history-enhanced data types must explicitly prove linearizability between the two histories.

The proofs here involve showing that each process is a *non-atomic refinement* [Derrick and Wehrheim 2003; 2005] of the abstract data type. To cope with incomplete histories, Derrick et al. use an additional set R that stores a set of return events for pending invocations that have linearized but not yet returned, and therefore, contributes to

the operation in the corresponding abstract history hs. In particular, assuming $\mathrm{bseq}(X)$ denotes bijective sequences generated from a set X, some $h_0 \in \mathrm{bseq}(R)$ can be used as the h_0 that completes pending invocations. The set $\mathrm{bseq}(R)$ contains all sequences constructed from R, so that each element of R appears in the sequence exactly once.

The proofs refer to the *status* (of type $STATUS ::= IDLE \mid IN\langle\langle V\rangle\rangle \mid OUT\langle\langle V\rangle\rangle$) of each process. Namely, process p has status $IDLE$ iff p is not executing any operation, $IN(x)$ iff p is executing an operation with input x, but has not passed the linearization point of the operation, and $OUT(r)$ iff p is executing an operation and has passed the linearization point with return value r. This is combined with a function $runs: CState \times P \to O \cup \{none\}$ denoting that the operation the given process is executing in a given state ($none$ if the process is idle), and a function $status: CState \times P \to STATUS$, which determines whether or not the process contributes a return event in a given state. The encoding of the *status* is such that its value is $IDLE$ if $runs(cs,p) = none$; is $IN(x)$ if $runs(cs,p) = op$ and $cs(pc(p))$ has executed the linearization point of op; and is $OUT(r)$ if $runs(cs,p) = op$ and $cs(pc(p))$ has executed the linearization point of op operation that returns value r.

The forward simulation relation rep is then of the following form, where $pi(n,h)$ denotes that $h(n)$ is a pending invocation event, i.e., $h(n)$ is an invocation and for all $m > n$, $h(m)$ is not a return event that matches $h(n)$.

$$
\begin{aligned}
&rep((as,hs),(cs,h)) \mathrel{\widehat{=}} \\
&\quad ABS(as,cs) \wedge INV(cs) \wedge (\forall p,q \bullet p \neq q \Rightarrow D(cs,p,q)) && (5)\\
&\quad \wedge\ (\forall n \bullet pi(n,h) \Rightarrow runs(cs,\pi_1(h(n))) = \pi_2(h(n))) && (6)\\
&\quad \wedge\ \forall p,x \bullet status(cs,p) = IN(x) \Rightarrow \exists n \bullet pi(n,h) \wedge h(n) = inv(p,runs(cs,p),x) && (7)\\
&\quad \wedge\ \exists R \bullet R = \{ret(p,op,r) \mid runs(cs,p) = op \wedge status(cs,p) = OUT(r)\} && (8)\\
&\qquad\quad \wedge\ \forall h_0 : \mathrm{bseq}(R) \bullet linearizable(h,h_0,hs)
\end{aligned}
$$

Here, (5) states that both abstraction ABS and invariant INV hold, and that $D(cs,p,q)$ holds, which ensures interference freedom for the local states of process p are not modified by execution of process q. Conjunct (6) states that if $h(n)$ is a pending invocation, then function $runs$ is accurate. Conjunct (7) states that whenever process p's status is $IN(x)$ for some x, there must exist an index $n \in \mathrm{dom}(h)$ such that $h(n)$ is a pending invocation, and corresponds to an invocation that is executing $runs(cs,p)$ with input x. Finally conjunct (8) relates h to hs using the set of processes with status OUT. It requires that there exist a set R of events corresponding to processes that have executed a linearizing statement, but not yet returned, such that for any bijective sequence h_0 generated from R, $linearizable(h,h_0,hs)$ holds.

Using rep, a number of process-local[7] proof obligations that do not need to refer to histories hs and h are generated, and a theorem that ensures satisfaction of the process-local properties that implies rep is proved. These proof obligations use the *status* function to determine the correct proof rule to apply. For example, the proof obligation below is for steps of process p that transition from a status $IN(in)$, where COp_p potentially corresponds to the execution of a linearization point.

$$
\begin{aligned}
&\forall as: AS, cs, cs': CState, p: P \bullet rep(as,cs) \wedge status(cs,p) = IN(in) \wedge COp_p(cs,cs') \Rightarrow \\
&\quad status(cs',p) = IN(in) \wedge rep(as,cs') \vee \\
&\quad (\exists as', out \bullet AOp_p(in,as,as',out) \wedge status(cs',p) = OUT(out) \wedge rep(as',cs'))
\end{aligned}
$$

Verifying invocation and response transitions are straightforward because the corresponding abstract state representation is not modified, and stuttering transitions are

[7] Thread-local in the terminology of Derrick et al. [2011a].

straightforward because neither the histories nor the state representations are modified. The non-stuttering transitions linearize the abstract object. This is reflected in the *status* function, whose value changes from $IN(x)$ before the transition to $OUT(r)$ after the transition. Locality of the proof method is guaranteed using the well-established technique of non-atomic refinement [Derrick and Wehrheim 2005] (we refer the interested reader to [Derrick et al. 2011a] for details).

4.4. Method 3: Augmented states

The method of Vafeiadis [2007], requires the following steps.

(1) Introduce auxiliary variables to the existing program, at least one of which is an abstraction of the data type in question, then define the abstract operations on these auxiliary variables that are required to be implemented by the concrete program.
(2) Identify the linearization points of the concrete implementation, then introduce the appropriate auxiliary statements at each linearization point.
(3) Define a rely condition by identifying statements that modify the global state, and developing an abstraction of each statement. The overall rely is a disjunction of each such abstraction.
(4) Define and prove an invariant that links the abstract and concrete representations.

Vafeiadis' proofs are performed using the RGSep framework [Vafeiadis 2007; Vafeiadis and Parkinson 2007]. In this paper, for uniformity, we translate the example expressed in RGSep into Z.

For the `add` operation, a state space is extended with a fresh variables `AbsRes` (representing the abstract result) and `S` (representing the abstract set) to obtain an augmented state *AugState*. In addition, the fixed linearization points `A5` and `A7` are augmented as follows, where the brackets `< >` delimit atomicity.

```
add(x):
 ...
 A5: <n1.next := n2; AbsRes := (x ∉ S); S := (S ∪ {x})>
 ...
 A7: <res := false; AbsRes := (x ∉ S); S := (S ∪ {x})> ...
```

Note that at `A7`, the auxiliary code sets `AbsRes` to *false* (i.e., `x` $\notin$ `S`), and therefore the abstract set `S` remains unchanged. The `remove` operation is similar, therefore its details are elided.

Assume that $lock(n)$ returns the id of the process that currently holds the lock on node n and that $lock(n) = \varnothing$ holds if no process has locked n. Furthermore, assuming that $val(n)$, $next(n)$ and $mark(n)$ denote the value, next and mark fields of n, respectively, we define predicates:

$$lvn(n) \mathrel{\widehat=} (lock(n), val(n), next(n))$$
$$lvnm(n) \mathrel{\widehat=} (lock(n), val(n), next(n), mark(n))$$

The non-stuttering actions of a program's environment are abstracted by rely conditions, which are relations on the pre-post states, representing transitions that modify the global state[8]. We replace all instances of separating conjunction '$*$' by logical conjunction '$\wedge$', which enables simpler comparison among the different methods. We discuss the differences that arise from this translation where needed.

$$AugState \mathrel{\widehat=} CState \wedge [S : \mathbb{P}\, V]$$

[8]Vafeiadis [2007] defines provisos for some of these actions, which we interpret as preconditions of each Z operation.

$$
\begin{aligned}
Lock_p &\mathrel{\widehat{=}} [\Delta AugState, n{:}Node \mid lock(n) = \varnothing \land p \neq 0 \land lock'(n) = \{p\}] \\
Unlock_p &\mathrel{\widehat{=}} [\Delta AugState, n{:}Node \mid lock(n) = \{p\} \land p \neq 0 \land lock'(n) = \varnothing] \\
Mark_p &\mathrel{\widehat{=}} \left[\Delta AugState, n, n1{:}Node, v{:}V \;\middle|\; \begin{array}{l} lvn(n1) = (p, v, n) \land \\ lvnm'(n1) = (p, v, n, true) \land \\ S' = S \backslash \{v\} \end{array}\right] \\
Add_p &\mathrel{\widehat{=}} \left[\begin{array}{l}\Delta AugState, \\ n1, n2, n3{:}Node, u, v{:}Val\end{array} \;\middle|\; \begin{array}{l} (u < v < val(n3)) \land lvn(n1) = (p, u, n3) \land \\ lock(n3) = \{p\} \land lvn(n2) = (\varnothing, v, n3) \land \\ next'(n1) = n2 \land S' = S \cup \{v\} \end{array}\right] \\
Remove_p &\mathrel{\widehat{=}} \left[\begin{array}{l}\Delta AugState, \\ n1, n2, n3{:}Node, u, v{:}V\end{array} \;\middle|\; \begin{array}{l} lvn(n1) = (p, u, n2) \land \\ lvnm(n2) = (p, v, n3, true) \land \\ lvn'(n1) = (p, u, n3) \end{array}\right]
\end{aligned}
$$

The rely condition for process p is

$$Rely_p \mathrel{\widehat{=}} \bigvee_{q \in P \backslash \{p\}} Lock_q \lor Unlock_q \lor Add_q \lor Mark_q \lor Remove_q$$

which describes the potential global modifications that the environment of process p can make. With this encoding, one can clearly see that the rely condition is an abstraction of statements of `add` and `remove` that modify the global state.

Vafeiadis [2007] requires annotation of code using separation logic-style assertions. In addition, building on the framework of Jones [1983], these assertions must be stable with respect to the rely conditions. The proof outlines for the lazy set are elided by Vafeiadis [2007], however, may be reconstructed from the other list examples in the thesis. We further adapt the proof outlines using Z-style notation. The invariants are formalised using the following predicates, where $ls(x, A, y)$ converts the linked list from node x to node y into an sequence A (where we assume y is reachable from x), predicate $sorted(A)$ holds iff A is sorted in ascending order, and $s(A)$ returns the set of elements corresponding to A.

$$
\begin{aligned}
ls(x, A, y) &\mathrel{\widehat{=}} (x = y \land A = \langle\rangle) \lor \\
&\quad \exists v, z, B \bullet x \neq y \land A = \langle v \rangle \frown B \land val(x) = v \land next(x) = z \land ls(z, B, y) \\
sorted(A) &\mathrel{\widehat{=}} \textbf{if } A \in \{\langle\rangle, \langle a \rangle\} \textbf{ then } true \\
&\quad \textbf{elseif } A = \langle a, b \rangle \frown B \textbf{ then } (a < b) \land sorted(\langle b \rangle \frown B) \textbf{ else } false \\
s(A) &\mathrel{\widehat{=}} S = \mathrm{ran}(A) \backslash \{-\infty, \infty\}
\end{aligned}
$$

Note that due to a typographical error, the case of the `add` operation that returns *false* is missing in [Vafeiadis 2007], however, it can be reconstructed from the `remove` operation (see Fig. 12). Of course, the annotations in Fig. 12 are not available in Z, but can be encoded as invariants on the overall specification by explicitly introducing a program counter variable. For example, given that $pc(p)$ denotes the program counter for process p, whose value is a program label, the assertion at `A7` can be encoded as a predicate:

$$
\begin{aligned}
POA7_p \mathrel{\widehat{=}} pc(p) = \texttt{A7} \Rightarrow \exists n, A, B \bullet ls(Head, A, n3_p) \land lvn(n3_p) = (p, x, n) \land \\
ls(n, B, Tail) \land s(A \frown \langle x \rangle \frown B)
\end{aligned}
$$

Such proof obligations must be resilient to interference from other processes [Owicki and Gries 1976], and hence, one must verify that the following holds for each $p, q \in P$ such that $p \neq q$, and $Env_q \in \{Lock_q, Unlock_q, Mark_q, Add_q, Remove_q\}$.

$$POA7_p \fatsemi Env_q \Rightarrow POA7'_p$$

Liang and Feng [2013a] provide outlines for the `remove` and `contains` operations albeit using a different framework, and define a number of additional predicates prior

```
add(x) :
```
...
$$\left\{\begin{array}{l}\exists u, v \bullet \exists n, A, B \bullet ls(Head, A, n1_p) \wedge lvn(n1_p) = (p, u, n3_p) \\ \qquad\qquad \wedge lvn(n3_p) = (p, v, n) \wedge ls(n, B, Tail) \wedge s(A \frown \langle u, v \rangle \frown B) \\ \quad \wedge lvn(n2_p) = (\varnothing, x, n3_p) \wedge u < x \wedge x < v \end{array}\right\}$$

```
A5: <n1.next := n2; AbsRes := (x ∉ S); S := (S ∪ {x})>
```
$$\left\{\begin{array}{l}\exists u, v \bullet \exists n, A, B \bullet ls(Head, A, n1_p) \wedge lvn(n1_p) = (p, u, n2_p) \\ \qquad\qquad \wedge lvn(n2_p) = (p, x, n) \wedge ls(n, B, Tail) \wedge s(A \frown \langle u, x \rangle \frown B) \end{array}\right\}$$
...
$$\left\{\exists n, A, B \bullet ls(Head, A, n3_p) \wedge lvn(n3_p) = (p, x, n) \wedge ls(n, B, Tail) \wedge s(A \frown \langle x \rangle \frown B)\right\}$$
```
A7: <res := false; AbsRes := (x ∉ S); S := (S ∪ {x})>
```
$$\{\exists A \bullet ls(Head, A, Tail) \wedge s(A)\}$$
...

Fig. 12. Reconstructed proof outline for `add(x)`

to the proof for `remove` and `contains`. These predicates largely mimic Vafeiadis' rely conditions. As with Vafeiadis' proofs, a translation of Liang and Feng's formalisation to Z is also possible, but due to the similarities between the proof methods, we elide the details in this survey and refer the interested reader to Liang and Feng [2013b].

4.5. Discussion

With the advances in linearizability verification, correctness of the optimistic set is straightforward, and there is even the possibility of automating the verification process (e.g., by extending methods of Vafeiadis [2010] and Dragoi et al. [2013][9]). We have presented a detailed account of three methods that manually identify the linearization points, as well as abstraction relations and invariants. These methods are based on differing formal foundations: method 1 uses I/O Automata, method 2 uses Z, and method 3 uses RGSep. To simplify comparison between these approaches, we have translated each of these to Z. An advantage of RGSep (method 3) that is lost in the translation to Z is the ability to syntactically distinguish between predicates that may be affected by the environment. However, as already discussed, the majority of predicates in each assertion are non-local, and hence, the loss of this feature does not overly affect the complexity of the proof. Methods 1 and 2 are mechanised in theorem provers PVS and KIV, respectively. Tool support for extensions to method 3 have been developed, and there is a possibility for mechanising proofs using method 3 directly, but this has thus far not been done. Each of the methods supports process-local verification. Method 1 proves invariants that describe the behaviours of the other processes, method 2 explicitly encodes interference freedom conditions in the refinement relation, and method 3 additionally supports compositionality via rely-guarantee reasoning.

The underlying challenges in verifying linearizability are manifested in each of the proof methods in essentially the same way. Namely, the identification of the correct abstraction relations and invariants, correct identification of linearization points and

[9]Note that the optimistic set in [Vafeiadis 2010] does not use a marking scheme, and hence, is different from the algorithm in Section 4.1. The methods in [Dragoi et al. 2013] can only automatically verify algorithms with helping mechanisms.

```
contains(x) :
C1: curr := Head;
C2: while curr.val < x do
C3:   curr := curr.next
C4: res := !curr.mark and
          (curr.val = x)
C5: return res
```

Fig. 13. The `contains` operation

```
contains(x) :
...
...
C4a: r1 := !curr.mark;
C4b: res := r1 and
            (curr.val = x);
...
```

Fig. 14. Splitting atomicity (a)

```
contains(x) :
...
...
C4c: r1 := (curr.val = x);
C4d: res := !curr.mark and
                r1;
...
```

Fig. 15. Splitting atomicity (b)

the corresponding abstract changes that occur at each linearization point. These also remain the difficult aspects of a proof to automate.

5. CASE STUDY 2: A LAZY SET ALGORITHM

In this section, we present the full lazy set algorithm, which consists of a `contains` operation in addition to the `add` and `remove` operations of the optimistic set from Section 4. Section 5.1 presents the `contains` operation in detail. Despite the simplicity of this operation, its verification introduces significant complexity in the proof methods, requiring the use of more advanced verification techniques. Sections 5.2 and 5.3 present simulation-based proof methods with respect to canonical and sequential abstract specifications, respectively, and Section 5.4 presents a method based on augmented states.

5.1. The `contains` operation

A process executing `contains(x)` traverses the list (ignoring locks) from *Head*, stopping as soon as a node with value greater or equal to `x` is found. Value `true` is returned if the node is unmarked and its value is equal to `x`, otherwise `false` is returned. Unlike `locate`, the `contains` operation does not acquire locks, and performs at most one traversal of the linked list.

When verifying linearizability of the `contains` operation, atomicity constraints of an implementation often dictate that the expression in `C4` be split. Because the order in which the variables within a non-atomic expression are accessed is not known, there are two possible evaluation orders: Fig. 14 and Fig. 15, both of which use a temporary variable `r1`. To verify linearizability of the original operation in Fig. 13, both orders of evaluation must be verified. However, Derrick et al. [2011b] and Vafeiadis [2007] only consider the variation in Fig. 14, while Colvin et al. [2006a] only consider Fig. 15. It is also possible to consider both possibilities at the same time using logics that enable reasoning about the non-determinism in expression evaluation under concurrency [Hayes et al. 2013], which is the approach taken by Dongol and Derrick [2013].

Unlike the `add` and `remove` operations, none of the statements of `contains` qualify as valid linearization points. To see this, we consider the two most suitable candidates, i.e., `C4a` and `C4b`, and present counter-examples to show that neither of these are valid. The essence of the issue is that a verifier must decide whether or not the `contains` will return *true* or *false* (i.e., as its future behaviour) by considering the state of the shared object when `C4a` or `C4b` is executed, and this is impossible. Suppose `C4a` is chosen as the linearization point of the `contains` operation. Now consider the state of the shared linked list in Fig. 16 (a), where process p is executing `contains(50)` and has just exited its loop because $curr_p.val \geq 50$, but has not yet executed statement `C4a`. Suppose another process q executes `add(50)` to completion. This results in the linked list in Fig. 16 (b), which corresponds to an abstract state $\{3, 18, 50\}$. Execution of `C4a` by process p from this state will set $r1_p$ to *false*, and hence the `contains(50)` will return

false, even though the element 50 is in the set (corresponding to the shared linked list) when `C4a` is executed.

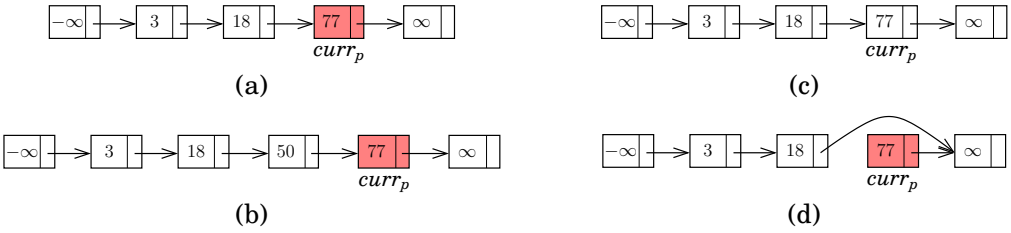

Fig. 16. Counter-examples for `C4a` and `C4b` as linearization points of `contains`

Similarly, suppose `C4b` is chosen to be the linearization point of the `contains` operation. Assume there are no other concurrent operations and that process p is executing `contains(77)` on the linked list in Fig. 11 (a), and execution has reached (but not yet executed) statement `C4b`. This results in the state of the linked list in Fig. 16 (c). Suppose another process q executes a `remove(77)` operation to completion. This results in Fig. 16 (d), corresponding to the abstract queue $\{3, 18\}$. Now, when process p executes `C4b`, it will set res_p to *true*, and hence, return *true* even though 77 is not in the abstract set corresponding to the shared linked list when `C4b` is executed. Therefore, neither `C4a` nor `C4b` are appropriate linearization points for `contains` .

Proving linearizability it turns out must consider the execution of other operations, i.e., the linearization point cannot be determined statically by examining the statements within the `contains` operation alone. Here, `contains` may be linearized by the execution of an `add` or a `remove` operation. As Colvin et al. point out:

> The key to proving that [Heller et al's] lazy set is linearizable is to show that, for any failed `contains(x)` operation, `x` is absent from the set at some point during its execution [Colvin et al. 2006a].

That is, within any interval in which `contains(x)` executes and returns *true*, there is some point in the interval such that the abstract set corresponding to the shared linked list contains x. Similarly, if `contains(x)` returns *false*, there is some point in the interval of execution such that the corresponding abstract set does not contain x. The statement that removes x from the set is also responsible for linearizing any `contains(x)` operations that may return false.

From a refinement perspective, the abstract specification resolves its nondeterminism earlier than the concrete implementation, resulting in a future concrete transition that cannot be matched with an abstract transition when the forward simulation rule (2) is used. Instead proofs must be performed using *backward simulation* [de Roever and Engelhardt 1996], which for a non-stuttering transition generates a proof obligation of the form:

$$\forall p : P \bullet COp_p \fatsemi rep \subseteq rep \fatsemi AOp_p$$

This states that, for any process p, if COp_p can transition from τ to τ' and τ' is related by *rep* to some abstract state σ', then there must exist an abstract state σ such that *rep* holds between τ and σ and AOp_p can transition from σ to σ'. Such proofs involve reasoning from the end of computation to the start, and hence, are more complicated than forward simulation. Equivalent to this is an encoding using prophecy variables [Abadi and Lamport 1991; Vafeiadis 2007; Zhang et al. 2012].

5.2. Method 1: Proofs against canonical automata

Colvin et al. split their simulation proofs by introducing an intermediate specification, that "eliminates the need to know the future" [Colvin et al. 2006a, pg 481]. They then prove backward simulation between the canonical and intermediate specifications and forward simulation between the intermediate and concrete specifications. To simplify the backward simulation, the intermediate specification is kept as similar to the canonical abstraction as possible, and almost no data refinement is performed.

The intermediate state introduces a local boolean variable $seen\_out(p)$ that holds for a process p executing `contains(x)` iff `x` has been absent from the abstract set since p invoked `contains(x)`. The invocation of the intermediate `contains(x)` operation sets $seen\_out(p)$ to *false* if x is in S and to *true* otherwise. Furthermore, when the linearization point of `remove(x)` is executed, in addition to linearizing itself, the executing process also linearizes all invoked `contains(x)` operations that have not yet set their $res(p)$ value. Therefore, $IContInv_p$ (which invoke the `contains`) and $IRemOK_p$ (which performs the main remove operation) in the intermediate specification are defined as follows:

$$IContInv_p(x) \mathrel{\widehat{=}} [\Delta IState, x?: V \mid ContInv_p(x?) \land seen\_out'(p) = (x? \notin S)]$$

$$IRemOK_p(x) \mathrel{\widehat{=}} \left[\Delta IState, x?: V \;\middle|\; \begin{array}{c} RemOK_p(x?) \land \\ \forall q: P \bullet pc(q) = CIn \land v(q) = v(p) \Rightarrow \\ seen\_out'(q) = true \end{array}\right]$$

The intermediate `contains(x)` operation is allowed to return *false* whenever $seen\_out(x)$ holds, therefore, $ContFail_p$ is replaced by $IContFail_p$ below:

$$IContFail_p \mathrel{\widehat{=}} [\Delta IState \mid pc(p) = CIn \land seen\_out(p) \land res'(p) = false \land pc'(p) = COut]$$

Unlike $ContFail_p$, schema $IContFail_p$ can set $res(p)$ to *false* even if $v(p) \in S$ holds in the current state. This is allowed because whenever $pc(p) = CIn \land seen\_out(p)$ holds, a state for which $v(p) \notin S$ holds must have occurred at some point since the invocation of the `contains` operation. When $pc(p) = CIn \land seen\_out(p) \land v(p) \in S$ holds, both $IContOK_p$ and $IContFail_p$ are enabled and process p may non-deterministically choose to match with $res(p) = true$ (in the current state) or with $res(p) = false$ (having linearized at some point in the past).

The backward simulation relation *bsr* below between the canonical and intermediate state spaces is relatively straightforward. In particular, one obtains the following for an intermediate state is and abstract state as.

$$bsr(is, as) \mathrel{\widehat{=}} is(S) = as(S) \land \forall p \bullet is(pc(p)) = as(pc(p)) \lor \begin{pmatrix} is(pc(p)) = CIn \land is(seen\_out(p)) \land \\ as(pc(p)) = COut \land as(res(p)) = false \end{pmatrix}$$

The second disjunct within the universal quantification is needed because p may have already executed $ContFail_p$ at the abstract level, and decided to return *false*, whereas the corresponding intermediate operation has not yet made its choice. Note that this delay in the intermediate specification is only allowed if $seen\_out(p)$ holds in the intermediate state.

A forward simulation is then used to prove refinement between the intermediate and concrete systems. As in the proof of the optimistic set (Section 4.2), this proof is simplified by introducing an auxiliary set aux_S to the concrete state space, which is updated in the same way as in Section 4.2. The proof also uses the same simulation as Section 4.2, but additional invariants related to the `contains` operation must be introduced. For example:

$$\forall p \bullet \begin{pmatrix} (pc(p) = 4c \land val(curr(p)) \neq v(p)) \lor \\ (pc(p) \in \{4c, 4d\} \land mark(curr(p))) \end{pmatrix} \Rightarrow seen\_out(p) \tag{9}$$

$$\forall p \bullet pc(p) = 4d \wedge \neg mark(curr(p)) \;\Rightarrow\; v(p) \in aux\_S \tag{10}$$

By (9), if the concrete program is in a position to return *false*, it must have already seen that the value being searched is not in the set, and by (10), if the concrete program is in a position to return *true*, the value being searched must be in the set. The proof of forward simulation then proceeds in a standard manner.

5.3. Method 2: Proofs against sequential specifications

This section summarises the proof by Derrick et al. [2011b], where linearizability is proved against a sequential set specification. The abstraction of the `contains` operation is therefore given by

$$AbsCont_p \mathrel{\widehat=} [\Delta ASState, x?: V, r!: \mathbb{B} \mid r! = (x? \in S)]$$

To cope with the non-determinism in the linearization points, Derrick et al. generalise the notion of a status by introducing $INOUT(in, out)$ that covers a situation in which an operation has *potentially* linearized with input *in* and output *out*. Thus, in this new setting:

$$STATUS ::= IDLE \mid IN\langle\langle V \rangle\rangle \mid OUT\langle\langle V \rangle\rangle \mid INOUT\langle\langle V \times V \rangle\rangle$$

For example, in the lazy set, a status $INOUT(3, true)$ denotes a process that has potentially executed its linearization point, with 3 as input and is set to return output *true*.

The proof proceeds by defining the status of each process in the concrete states. Namely, for a `contains(x)` operation executed by process p, given that *cs* is a concrete state, we assume that *status* is defined such that the following holds:

$$\begin{aligned}
(cs.pc(p) = C1 &\;\Rightarrow\; status(cs, p) = IN(x)) \wedge \\
(cs.pc(p) \in \{C2, C3, C4a\} &\;\Rightarrow\; status(cs, p) = INOUT(x, x \in cs.aux_S)) \wedge \\
(cs.pc(p) = C4b &\;\Rightarrow\; status(cs, p) = OUT(cs.(val(curr(p))) = x)) \wedge \\
(cs.pc(p) = C5 &\;\Rightarrow\; status(cs, p) = OUT(cs.res(p)))
\end{aligned}$$

While executing $C2$, $C3$ or $C4a$, a `contains` operation may now "change its mind" about the linearization point and its eventual return value as often as necessary, provided each change is justified by the current set representation. In particular, a process q marking the element that is searched by process p will change the status of process p executing `contains` to *false*. This is justified because the value being searched by p is also removed from the set representation. A process q inserting a node with x after node $curr(p)$ will change p's status to *true*, which is justified because x is also added to the set representation.

To cope with the fact a step in an operation potentially linearizes those in (several) other operations, two new simulation types are introduced in addition to those in Fig. 5 (see [Derrick et al. 2011b] for full details). The left diagram of Fig. 17 shows the case where the execution of operation COp_p definitely sets its own as well as the linearization point of process q that executes an operation that does not modify the global state (e.g., a `contains` operation). The right hand side depicts the case where the abstract operation of process p is a potential linearization point for p that does not modify the abstract state (e.g,. a `contains` operation).

5.4. Method 3: Augmented states

The method of Vafeiadis also requires substantial changes to cope with verification of the `contains` operation. In particular, auxiliary statements that are able to linearize the currently executing `contains` operations must be introduced to the remove operation.

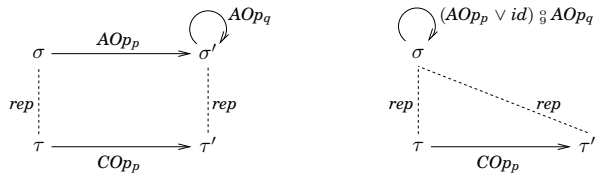

Fig. 17. Additional simulation types

The augmented state introduces a further auxiliary variable $OSet \subseteq P \times V \times \mathbb{B}$, where $(p, v, r) \in OSet$ iff process p is executing a `contains` operation with input v that has set its return value to r. This requires modification of environment actions that modify the shared state space. Operations $Lock_p$, $Unlock_p$, Add_p and $Remove_p$ are as given in Section 4.4. The $Mark_p$ action, which is an environment action for process p that marks a node, must also modify the abstract set S (as in Section 4.4) and the auxiliary $OSet$. In addition to setting the marked value to *true* and removing v from the abstract set, the executing process p also sets the return value of all processes in $C \subseteq OSet$ that are currently executing a `contains(v)` to *false*, which linearizes each of the processes in C.

$Mark_p$

$\Delta AugState$
$n, n1 : Node, B, C : (P \times V \times \mathbb{B}), v : V, r : \mathbb{B}$

$lvn(n1) = (p, v, n) \wedge OSet = B \cup C \wedge (\forall b : B \bullet \pi_2(b) \neq v) \wedge (\forall c : C \bullet \pi_2(c) = v)$
$lvnm'(n1) = (p, v, n, true) \wedge S' = S \backslash \{v\}$
$OSet' = B \cup \{(q, v, false) \mid \exists r \bullet (q, v, r) \in C\}$

In addition, two environment steps that add and remove triples of type $P \times V \times \mathbb{B}$ to/from the auxiliary variable $OSet$ are introduced. These represent environment processes that invoke and complete a `contains` operation.

$$AddOut_p \mathrel{\widehat=} [\Delta AugState, v : V, r : \mathbb{B} \mid (\forall o : OSet \bullet o.1 \neq p) \wedge OSet' = OSet \cup \{(p, v, r)\}]$$
$$RemOut_p \mathrel{\widehat=} [\Delta AugState, v : V, r : \mathbb{B} \mid (p, v, r) \in OSet \wedge OSet' = OSet \backslash \{(p, v, r)\}]$$

The auxiliary code to the `add` and `remove` operations are as before, but a `remove(x)` operation must additionally linearize processes in `OSet` that are executing `contains(x)`. Thus, statement `R3` is augmented as follows:

```
remove(x):
 ...
 R3:   <n2.mark := true; AbsRes(this) := (x ∈ S);
        for each q ∈ OSet do if q.2 = n2.val then AbsRes(q) := false >
...
```

The augmented version of the `contains` operation is given below[10]. Like Vafeiadis [2007], details of the annotation for the proof outline are elided below, but the interested reader may consult Liang and Feng [2013a].

[10]The presentation by Vafeiadis [2007] suffers from a few typos, which are confirmed by the proof by Liang and Feng [2013a]. In particular, the auxiliary code that linearizes itself (in statement `C4a`) should only set the abstract result to true if both `not res` and `curr.val = e` hold, as opposed to only `not res` as indicated by Vafeiadis [2007].

```
contains(x) :
     <AbsRes(this) := (x ∉ S); OSet := OSet ∪ {this}>;
 C1: curr := Head;
 ...
C4a: <r1 := curr.marked;
      AbsRes(this) := (not r1 and curr.val = x);
      OutOps := OutOps \ {this}> ...
```

The augmentation is such that any process p that invokes `contains(x)` initially linearizes to *true* or *false* depending on whether or not `x` is in the abstract set, then records itself in `OSet`. This allows other processes executing `remove(x)` to set p's linearization point when `x` is marked (i.e., logically removed). The linearization point for an execution that returns *true* is set at statement `C4a` if $curr(p)$ points to an unmarked node with value x.

5.5. Discussion

The lazy set represents a class of algorithms that can only be verified by allowing an operation to set the linearization point of another, and its proof is therefore more involved. The methods we have considered tackle the problem using seemingly different techniques. However, translating each proof to a uniform framework, in this case Z, one can see that the underlying ideas behind the methods are similar, and experience in verification using one of these methods can aid in the proof in another. Identifying the linearization points and understanding the effects of linearization on object at hand remains the difficult task. Here, further complications arise because external operations potentially set the linearization point of the current operation.

Dongol and Derrick [2013] present a method for verifying linearizability using an interval-based framework, which aims to capture the fact that operations like `contains` must only observe the value being checked as being in the set at some point within its interval of execution. The logic is able to prove properties of the form

$$beh_p(\texttt{contains}(x), true) \wedge rely_p \;\Rightarrow\; \diamondsuit(x \in absSet)$$
$$beh_p(\texttt{contains}(x), false) \wedge rely_p \;\Rightarrow\; \diamondsuit(x \notin absSet)$$

Here, $beh_p(\texttt{contains}(x), true)$ defines an interval-based semantics of the behaviour of $\texttt{contains}(x)$ executed by process p that returns *true*, $rely_p$ is an interval predicate that defines the behaviour of the environment of p and $\diamondsuit(x \in absSet)$ is an interval predicate that holds if $x \in absSet$ is true at some point in the given interval. Such proofs allow one to avoid backward reasoning because the entire interval of execution is taken into account.

6. CASE STUDY 3: THE HERLIHY-WING QUEUE

We now discuss the third type of algorithm, where none of the atomic program statements qualify as linearization points. Instead, execution of an atomic statement that linearizes an operation depends on future executions, and in addition, the potential linearization points may modify the representation of the data structure at hand. One such algorithm is the array-based queue by Herlihy and Wing [1990], which we present in Fig. 18. The abstract object corresponding to a concrete state cannot be determined by examining the shared data structure (in this case a shared array) alone — one must additionally take into consideration the currently executing operations and their potential future executions. As these operations may potentially modify the shared data structure in the future, each concrete state ends up corresponding to a set of abstract states.

```
enq(lv : V)
 E1: (k,back) := (back, back+1); // increment
 E2: AR[k]:= lv; // store
 E3: return

deq():
 D1: lback := back; k:=0; lv := null;
 D2: if k < lback goto D3 else goto D1
 D3: (lv, AR[k]) := (AR[k], lv); // swap
 D4: if lv != null then goto D6 else goto D5
 D5: k := k + 1; goto D2
 D6: return(lv)
```

Fig. 18. The Herlihy-Wing queue

In Fig. 18, each line corresponds to a single atomic statement, including for example D1, which consists of several assignments. These operations operate on an infinite array, AR (initially null at each index), and use a single shared global counter, back (initially 0) that points to the end of the queue.

An enqueue operation (enq) atomically increments back (line E1) and stores the old value of back locally in a variable k. Thus executing E1 allows the executing process to reserve the index of back before the increment as the location at which the enqueue will be performed. The enqueued value is stored at E2. A dequeue operation (deq) stores back locally in lback, then traverses AR from the front (i.e., from index 0) using k. As it traverses AR, it swaps the value of AR at k with null (D3). If a non-null element is encountered (D4), then this value is returned as the head of the queue. If the traversal reaches lback (i.e., the local copy of back read at line D1) and a non-null element has not been found, then the operation restarts. Note that deq is *partial* [Herlihy and Wing 1990] in that it does not terminate if AR is null at every index. In particular, a dequeue only terminates if it returns a value from the queue.

To see why verifying linearizability of the algorithm is difficult, we first show that neither E1 nor E2 qualify as a valid linearization points for enq. It is straightforward to derive a similar counter example for E3. Suppose E1 is picked as the linearization point and consider the following complete execution, where $p, q, r \in P$. Assume p and q enqueue v_1 and v_2, respectively.

$$\langle \mathtt{E1}_p, \mathtt{E1}_q, \mathtt{E2}_q, \mathtt{D1}_r, \mathtt{D2}_r, \mathtt{D3}_r, \mathtt{D4}_r, \mathtt{D5}_r, \mathtt{D2}_r, \mathtt{D3}_r, \mathtt{D4}_r, \mathtt{D6}_r, \mathtt{E2}_p, \mathtt{E3}_q, \mathtt{E3}_p \rangle \tag{11}$$

Although $\mathtt{E1}_p$ is executed before $\mathtt{E1}_q$, the dequeue operation returns v_2 before v_1, contradicting FIFO ordering, and hence, E1 cannot be a linearization point. Now suppose E2 is picked as the linearization point and consider the following complete execution:

$$\langle \mathtt{E1}_p, \mathtt{E1}_q, \mathtt{D1}_r, \mathtt{D2}_r, \mathtt{D3}_r, \mathtt{D4}_r, \mathtt{D5}_r, \mathtt{E2}_p, \mathtt{E2}_q, \mathtt{D2}_r, \mathtt{D3}_r, \mathtt{D4}_r, \mathtt{D6}_r, \mathtt{E3}_q, \mathtt{E3}_p \rangle \tag{12}$$

Now, $\mathtt{E2}_p$ is executed before $\mathtt{E2}_q$, but deq returns v_2 before v_1 has been dequeued.

The histories corresponding to both executions are however, linearizable because the operation calls $\mathtt{enq}_p$, $\mathtt{enq}_q$ and $\mathtt{deq}_r$ overlap, allowing their effects to occur in any order. In particular, both (11) and (12) correspond to history

$$\langle enq_p^I(v_1), enq_q^I(v_2), deq_r^I, deq_r^R(v_2), enq_q^R, enq_p^R \rangle$$

which is linearizable.

Aside from the proof sketch in Herlihy/Wing's original paper [Herlihy and Wing 1990], there are two known formal proofs of linearizability: [Schellhorn et al. 2012; 2014] (which uses backwards simulation) and [Henzinger et al. 2013b] (which decomposes the problem into several *aspects*). Henzinger et al.'s main ordering property uses

prophecy variables, and hence, must perform reasoning similar to backward simulation.

Backward simulation and prophecy variables are known to be equivalent formulations, that allow the future non-determinism to be taken into account [de Roever and Engelhardt 1996]. Both allow one to capture the fact that in order to decide whether the enqueue operation has taken effect, one must consider the state of all currently executing operations

6.1. Method 1: Backward simulation proofs

Schellhorn et al. [2014] have shown that backward simulation is sufficient for proving linearizability, i.e., backward simulation with the addition of auxiliary history variables is a complete method for proving linearizability. These methods however, do not show *how* such a simulation relation may be constructed, and hence, creativity is required on the part of the verifier to develop the correct simulation relation. As already discussed, each concrete state corresponds to multiple abstract queues depending on the states of the executing operations. Schellhorn et al's approach is to encode, within the simulation relation, all possible ways in which the currently executing `enq` operations can complete, as well as all possible ways in which these could be dequeued. To this end, they construct a so-called *observation tree*. In effect, this constructs the *set* of all possible queues that could relate to the current concrete queue based on the state of `AR` and the pending concurrent operations. The proof methods build on previous work on potential linearization points (Section 5.3), the difference here is that linearizing external operations modifies the data structure in question.

For example, statement `E1` of `enq` is a potential linearization point, and hence, one must perform case analysis to check whether or not its execution linearizes the currently executing enqueue operation. The non-linearizing cases are straightforward as one must only check that the set of queues from the post-state are the same as those in the pre-state. For the linearizing case, there must be some abstract queue related to the concrete post-state for which the element being enqueued is at the tail of some abstract queue related to the pre-state. Proving this is further complicated by the fact that an `enq` operation call that executes `E1` may 'overtake' other `enq` operation calls that executed `E1` earlier (and hence have a lower local value of `k`), causing the effect of a latter execution of `E1` to occur earlier. In fact, depending on the configuration of operation calls in the concrete state, executing `E1` may even overtake other `enq` operation calls that have executed `E2`.

The full argument is rather complex, and hence, we do not present further details of this verification here. Instead we ask the interested reader to consult [Schellhorn et al. 2014]. We note that their proofs are fully mechanised using the KIV theorem prover.

6.2. Method 2: Aspect-oriented proofs

A second proof of the Herlihy/Wing queue is given by Henzinger et al. [2013b], who define a set of *aspects* that characterise the behaviour of a queue and show that Herlihy/Wing's queue satisfies these aspects. In particular, the following aspects are required of a FIFO queue:

VFresh. A dequeue event returning a value not inserted by any enqueue event.
VRepet. Two dequeue events returning the value inserted by the same enqueue event.
VOrd. Two ordered dequeue events returning values inserted by enqueue events in the inverse order.
VWit. A dequeue event returning empty even though the queue is never logically empty during the execution of the dequeue event.

These aspects are only shown to be necessary and sufficient for proving linearizability if the implementation is *purely-blocking*, meaning that from any reachable state, any pending operation, if run in isolation will either terminate or its entire execution does not modify the global state.

For the Herlihy-Wing queue, VWit is irrelevant as the dequeue loop only terminates if a non-null element is read, i.e., it never returns empty. Both aspects VFresh and VRepet are straightforward to check. VOrd, however is more involved as it must reason about potential reordering of `enq` operation (as encountered by [Schellhorn et al. 2014]). Aspect VOrd is reformulated as POrd, which states the following.

> Fix a value v_2 and consider a history c where every method call enqueuing v_2 is preceded by some method call enqueuing some different value v_1 and there are no `deq()` calls returning v_1 (there may be arbitrarily many concurrent `enq()` and `deq()` calls enqueuing or dequeuing other values). The goal is to show that in this history, no `deq()` return v_2. [Henzinger et al. 2013b]

In other words, if an ordering of values v_1 and v_2 in a history c has been decided so that the enqueue of v_1 precedes the enqueue of v_2, and no dequeue operation calls return the first value v_1, then there are no operations that dequeue the second value v_2.

The proof POrd for the Herlihy/Wing queue requires the use of *prophecy variables* that allow dequeue operations to 'guess' the value that they will dequeue. Assertions on prophecy variables are encoded within the program code, then verification proceeds by showing that these guesses are correct. Again, we leave out the full details of the proof method, and ask the interested reader to consult [Henzinger et al. 2013b].

6.3. Discussion

The Herlihy-Wing queue represents a class of algorithms that can only be proved linearizable by considering the future behaviours of the currently executing operation calls, further complicated by the potential for these operations to modify the data structure at hand. Reasoning must therefore appeal to backward simulation or prophecy variables.

Schellhorn et al. [2014] use a monolithic backward simulation relation that captures all possible future behaviours at the abstract level. The method has been show to be complete for verifying linearizability, however, developing and verifying such a simulation relation is a complex task. The aspect-oriented proof method decomposes a linearizability proof for purely blocking algorithms into simpler aspects that are (in theory) easier to verify [Henzinger et al. 2013b]. However, it is currently not clear whether every data structure can be decomposed into such aspects.

These are not the only method capable of handling future linearization points — two other methods, both based on backward simulation, could be applied to verify the Herlihy-Wing queue. We have not presented a detailed comparison here as they have not verified the Herlihy-Wing queue (i.e., we do not attempt a proof using their methods ourselves). Groves et al's backward simulations against canonical automata can cope with future linearization points [Doherty et al. 2004a]. Tofan et al. [2014] have continued to improve the simulation-based methods (Section 6.1), and incorporated the core theory into a interval-based rely/guarantee framework. Here, linearizability is re-encoded using *possibilities*, which describe the orders of completions of pending operation calls. The use of symbolic execution simplifies mechanisation of their approach within KIV. Their methods have been applied to verify correctness of an array-based multiset with insert, delete, and lookup operations. An interesting aspect of this algorithm is that it is possible for a lookup of an element x to return *false* even if the element x is in the array in all concrete states throughout the execution of the lookup

operation. Their methods have been linked to the completeness results of Schellhorn et al. [2014].

7. CONCLUSIONS

There has been remarkable progress since Herlihy and Wing's original paper on linearizability [Herlihy and Wing 1990], and with the increasing necessity for concurrency, this trend is set to continue. The basic idea behind linearizability is simple, yet it provides a robust consistency condition applicable to a large number of algorithms, and in some cases precisely captures the meaning of atomicity [Raynal 2013]. For concurrent objects, linearizability has shown to coincide with contextual observational refinement [Filipović et al. 2010], ensuring that the behaviours of client objects are preserved. Linearizability is *compositional* in the sense that a set of objects is linearizable if each object in the set is linearizable [Herlihy and Shavit 2008; Herlihy and Wing 1990], making it an appealing property. Besides shared variable concurrent objects, linearizability has also been applied to distributed systems [Birman 1992], databases [Ramamritham and Chrysanthis 1992] and fault-tolerant systems [Guerraoui and Schiper 1996].

This paper considered verification of linearizability, and the associated proof methods that have been developed for it in the context of concurrent objects. Necessity of such proofs is alluded to by the subtleties in the behaviours of the algorithms that implement concurrent objects, and by the fact that it's errors have been found in algorithms that were previously believed to be correct [Doherty et al. 2004a; Colvin and Groves 2005]. Current proof techniques continue to struggle with the scalability and as a result, only a handful of fine-grained algorithms have been formally verified to be linearizable. The longest fully verified algorithm (in terms of lines of code) is the Snark algorithm [Doherty et al. 2004a]. However, number of lines of code is not an indicator of complexity, with even simple algorithms like Herlihy and Wing's queue [Herlihy and Wing 1990] posing immense challenges [Schellhorn et al. 2012; Schellhorn et al. 2014; Henzinger et al. 2013b] due to the fact that future behaviour must be considered.

Our survey has aimed to answer the questions that were posed in Section 1. We now return to these to discuss concluding remarks.

Locality of the proof method. Each of the methods we've considered enable localised reasoning, only requiring the behaviour of a single process to be considered. However, interference must be accounted for in the invariants and refinement relations generated, complicating each verification step. Namely, one must show that holds locally and is preserved by the each step of an arbitrarily chosen process, and that it holds in the presence of interference from other processes.

Compositionality of the proof method. Some methods have incorporated Jones-style rely/guarantee reasoning into their respective frameworks (e.g., RGSep and RGITL), allowing potential interference from the environment to be captured abstractly by a rely condition. An additional step of reasoning is required to show that the rely condition is indeed an abstraction of the potential interference, but once this is done, a reduction in the proof load is achieved via a reduction in the number of cases that must be considered.

Contribution of the underlying framework. None of the existing frameworks thus far provide a silver bullet for linearizability verification. Identification of the linearization points and appropriate representation relations remain the difficult aspects of a proof. If the verifier believes an algorithm to have fixed linearization points, then it would be fruitful to attempt an initial verification using a tool such as the one provided by Vafeiadis [2010]. For more complex algorithms, using a setup such as the one provided by Colvin et al. [2006a] would allow invariants to be model checked prior to

verification. On the other hand, Derrick et al. [2011a] have developed a systematic method for constructing representation relations, invariants and interference freedom conditions as well as proof obligations that enables process-local verification. Techniques specific to certain implementations (e.g., the Hindsight Lemma, aspect-oriented verification) enable some decomposition possibilities, but have not been generalised to cope with arbitrary implementations.

Algorithms verified. A survey of these has been given in Section 3.2. There exist several other algorithms in the literature whose linearizability has been conjectured, but not yet been formally verified. For the frameworks we've studied, the number of algorithms verified is however not a measure of it capabilities; rather it is whether the framework can handle complex algorithms with future linearization points such as the Herlihy/Wing queue.

The verifications thus far, have only considered linear (flat) data structures. Recently, more challenging structures such as SkipTries [Oshman and Shavit 2013] and binary search trees [Chatterjee et al. 2014] have been developed. Their linearizability has been informally argued, but not mechanically verified. It is not easy to know exactly how the proof complexity increases for such data structures, however, the complex nature of the underlying algorithm and the abstract representations suggest that the proofs will also be more complex.

Mechanisation. Many of the methods described in this paper have additional tool support that support mechanical validation of the proof obligations, reducing the potential for human error. In some cases, automation has been achieved, reducing human effort, but these are currently only successful for algorithms with fixed linearization points and a limited number of algorithms with external linearization points.

Completeness. Completeness of a proof method is clearly a desirable quality — especially for proofs of linearizability, which require considerable effort. Backwards simulation alone is known to be complete for verifying linearizability against an abstract sequential specification [Schellhorn et al. 2012; Schellhorn et al. 2014]. Furthermore, a combination of forwards and backwards simulations is known to be complete for data refinement [Lynch 1996; de Roever and Engelhardt 1996], and combining auxiliary and prophecy variables is known to be complete for reasoning about past and future behaviour [Abadi and Lamport 1991].

Completeness of a method does not guarantee simpler proofs, as evidenced by the maximal backwards simulation constructed by Schellhorn et al. [2012; 2014] to prove linearizability. The completeness results [Schellhorn et al. 2012; 2014] show that by using the global theory any linearizable algorithm can be proved correct. This shows that for every linearizable object, a backward simulation in between abstract and concrete specification can be found. This result does, however, not directly give one a way of constructing this backward simulation. This is common to all completeness results: they state the existence of a proof within a particular framework, but not the way of finding this proof. That such proofs can for individual instances indeed be found, is exemplified by the highly non-trivial case study of the paper.

Future directions. Despite the numerous advances in verification methodologies, formal correctness proofs of concurrent algorithms in a scalable manner remains an open problem. This in turn affects verification of specific properties such as linearizability. The rate at which new algorithms are developed far outpace the rate at which these algorithms are formally verified. However, as concurrent implementations become increasingly prevalent within programming libraries (e.g., `java.util.concurrent`) the need for formal verification remains important.

So what will future algorithms look like? To reduce sequential bottlenecks, there is no doubt that concurrent objects of the future will continue to become more sophisticated with more subtle (architecture-specific) optimisations becoming prevalent. Proving linearizability of such algorithms will almost certainly require consideration of some aspect of future behaviour. It is therefore imperative that verification techniques that are able to handle this complex class of algorithms continue to be improved. The frameworks themselves must continue to integrate the various methods for proof decomposition (e.g., Section 3.1). For example, Tofan et al. [2014] have developed a framework that combines interval temporal logic, rely/guarantee and simulation proofs. Further simplifications could be achieved by extending the framework with aspects of separation logic. In some cases, decomposition of a proof into stages, e.g., using reduction, or interval-based abstraction has been useful, where the decomposition not only reduces the number of statements that must be considered, but also transfers the algorithm from a proof that requires consideration of external linearization points to a proof with fixed linearization points. Until a scalable generic solution is found, it is worthwhile pursuing problem-specific approaches (e.g., [Henzinger et al. 2013b; Dragoi et al. 2013]).

Another avenue of work is proof modularisation. To explain this, consider the elimination queue [Moir et al. 2005b], which embeds an elimination mechanism (implemented as an array) on top of the queue by Michael and Scott [1996] (with some modifications). Although linearizability of Michael and Scott's queue is well studied, current techniques require the entire elimination queue data structure to be verified from scratch. Development of modular proof techniques would enable linearizability proofs to be lifted from low-level data structures to more complex (optimised) versions. New results such as parameterised linearizability [Cerone et al. 2014] suggest that modular concurrent objects and associated proof techniques will continue to evolve.

Next, we discuss some additional aspects surrounding correctness of concurrent objects.

Model checking. An important strand of research is model checking, which due to the finite nature of the state space searched is often not adequate for ensuring linearizability. This paper has focused on verification methods, and hence, a detailed comparison of model checking methods have been elided. Like Colvin et al. [2006a], we believe model checking can play a complementary role in verification, e.g., allowing invariants to be model checked prior to verification to provide assurances that they can be proved correct. Methods for model checking linearizability may be found in [Vechev et al. 2009; Friggens 2013; Liu et al. 2009; Liu et al. 2013]; a comparison of these techniques is beyond the scope of this survey.

Progress properties. In many applications, one must often consider the progress properties that an algorithm guarantees. Here, like safety, several different types of progress conditions have been identified such as *starvation freedom*, *wait freedom*, *lock freedom* and *obstruction freedom* (see [Herlihy and Shavit 2008; 2011; Dongol 2009; 2006; Tofan et al. 2010; Liang et al. 2013; Gotsman et al. 2009]). Progress properties are not the main focus of this paper, and hence, discussion of methods for verifying them have been elided. Nevertheless, they remain an important property to consider when developing algorithms.

Parameter passing. A deficiency in linearizability theory is that it assumes data independence between libraries and clients, and hence only admits pass-by-value parameter passing mechanisms. Real-world systems however, also allow data sharing between libraries and clients, e.g., via pass-by-reference mechanisms. Here, ownership transfer between shared resources may occur. To this end, Gotsman and Yang [2012;

2013] have extended linearizability theory to cope with parameter sharing between concurrent objects and its clients. Cerone et al. [2014] have further extended these results and defined *parameterised linearizability* that allows linearizable objects to be taken as parameters to form more complex linearizable objects.

Relaxing linearizability. The increasing popularity of multicore/multiprocess architectures, has led to an increasing necessity for highly optimised algorithms. Here, researchers are questioning whether linearizability is itself causing sequential bottlenecks, which in turn affects performance. Due to Amdahl's Law, it is known that if only 10% of a program's code remains sequential, then one can achieve at best a fivefold speedup on a 10-core machine, meaning at least half of the machine's capability is wasted [Shavit 2011; Moir and Shavit 2007]. As a result, Shavit [2011] predicts future systems will trend towards more relaxed notions of correctness.

To this end several conditions weaker than linearizability have been defined to allow greater flexibility in an implementation, e.g., *quasi-linearizability* [Afek et al. 2010], *k-linearizability* [Henzinger et al. 2013a], *eventual consistency* [Shapiro and Kemme 2009]. Part of the problem is that linearizability insists on *sequential consistency* [Lamport 1997; Herlihy and Shavit 2008], i.e., that the order of events within a process is maintained. However, modern processors use local caches for efficiency, and hence, are not sequentially consistent. Instead, they only implement *weak memory models* that allow memory instructions to be reordered in a restricted manner [Adve and Gharachorloo 1996]. Shavit [2011] purports *quiescent consistency*, which only requires the real-time order of operation calls to be maintained when the calls are separated by a period of quiescence (which is a period without any pending operation invocations). Unlike linearizability, quiescent consistency does not imply sequential consistency, and hence, can be applied to weak memory models [Smith et al. 2014]. As quiescent consistency is weak condition, more recent work has consider quantitative relaxations to bridge the gap between linearizability and quiescent consistency [Jagadeesan and Riely 2014]. Dongol et al. [2015] have recently developed a framework for formally studying these correctness conditions, including those conditions developed for weak memory.

Weakening correctness conditions however, does not mean that the algorithms become easier to verify and furthermore methods for verifying linearizability can be ported to weaker conditions (e.g., see [Derrick et al. 2014]). Therefore, techniques for simplifying linearizability proofs will not be in vain if in the future weaker conditions become the accepted standard.

Acknowledgements

We are indebted to our anonymous reviewers, whose comments have helped improve this paper. We also thank Lindsay Groves for his comments on an earlier version.

REFERENCES

M. Abadi and L. Lamport. 1991. The Existence of Refinement Mappings. *Theor. Comput. Sci.* 82, 2 (1991), 253–284.

J.-R. Abrial. 2010. *Modeling in Event-B - System and Software Engineering*. Cambridge Univ. Press. I–XXVI, 1–586 pages.

J.-R. Abrial and D. Cansell. 2005. Formal Construction of a Non-blocking Concurrent Queue Algorithm. *Journal of Universal Computer Science* 11, 5 (May 2005), 744–770.

S. V. Adve and K. Gharachorloo. 1996. Shared Memory Consistency Models: A Tutorial. *IEEE Comp.* 29, 12 (1996), 66–76.

Y. Afek, G. Korland, and E. Yanovsky. 2010. Quasi-Linearizability: Relaxed Consistency for Improved Concurrency. In *OPODIS (LNCS)*, C. Lu, T. Masuzawa, and M. Mosbah (Eds.), Vol. 6490. Springer, 395–410.

D. Amit, N. Rinetzky, T. W. Reps, M. Sagiv, and E. Yahav. 2007. Comparison Under Abstraction for Verifying Linearizability. In *CAV (LNCS)*, W. Damm and H. Hermanns (Eds.), Vol. 4590. Springer, 477–490.

S. Bäumler, M. Balser, F. Nafz, W. Reif, and G. Schellhorn. 2010. Interactive verification of concurrent systems using symbolic execution. *AI Commun.* 23, 2-3 (2010), 285–307.

S. Bäumler, G. Schellhorn, B. Tofan, and W. Reif. 2011. Proving linearizability with temporal logic. *Formal Asp. Comput.* 23, 1 (2011), 91–112.

K. P. Birman. 1992. Maintaining Consistency in Distributed Systems. In *Proceedings of the 5th ACM SIGOPS European Workshop: MPDSS (EW 5)*. ACM, New York, NY, USA, 1–6.

J. P. Bowen. 1996. *Formal Specification and Documentation Using Z: A Case Study Approach*. International Thomson Computer Press.

R. M. Burstall. 1974. Program Proving as Hand Simulation with a Little Induction. In *IFIP Congress*. 308–312.

A. Cerone, A. Gotsman, and H. Yang. 2014. Parameterised Linearisability. In *ICALP (2) (LNCS)*, J. Esparza, P. Fraigniaud, T. Husfeldt, and E. Koutsoupias (Eds.), Vol. 8573. Springer, 98–109.

B. Chatterjee, N. Nguyen Dang, and P. Tsigas. 2014. Efficient lock-free binary search trees. In *PODC*, M. M. Halldórsson and S. Dolev (Eds.). ACM, 322–331.

R. Colvin, S. Doherty, and L. Groves. 2005. Verifying Concurrent Data Structures by Simulation. *Electr. Notes Theor. Comput. Sci.* 137, 2 (2005), 93–110.

R. Colvin and L. Groves. 2005. Formal Verification of an Array-Based Nonblocking Queue. In *ICECCS* (16-20 June 2005). IEEE Computer Society, 507–516.

R. Colvin, L. Groves, V. Luchangco, and M. Moir. 2006a. Formal Verification of a Lazy Concurrent List-Based Set Algorithm. In *CAV (LNCS)*, T. Ball and R. B. Jones (Eds.), Vol. 4144. Springer, 475–488.

R. Colvin, L. Groves, V. Luchangco, and M. Moir. 2006b. PVS proofs for Lazy Set Algorithm. (2006). http://www.mcs.vuw.ac.nz/research/SunVUW/LazyListFiles/

W. P. de Roever, F. de Boer, U. Hannemann, J. Hooman, Y. Lakhnech, M. Poel, and J. Zwiers. 2001. *Concurrency Verification: Introduction to Compositional and Noncompositional Methods*. Cambridge University Press.

W. P. de Roever and K. Engelhardt. 1996. *Data Refinement: Model-oriented proof methods and their comparison*. Number 47 in Cambridge Tracts in Theor. Comp. Sci. Cambridge University Press.

J. Derrick, B. Dongol, G. Schellhorn, B. Tofan, O. Travkin, and H. Wehrheim. 2014. Quiescent Consistency: Defining and Verifying Relaxed Linearizability. In *FM (LNCS)*, C. B. Jones, P. Pihlajasaari, and J. Sun (Eds.), Vol. 8442. Springer, 200–214.

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

J. Derrick, G. Schellhorn, and H. Wehrheim. 2011b. Verifying Linearisability with Potential Linearisation Points. In *FM (LNCS)*, M. Butler and W. Schulte (Eds.), Vol. 6664. Springer, 323–337.

J. Derrick and H. Wehrheim. 2003. Using coupled simulations in non-atomic refinement. In *ZB 2003: Formal Specification and Development in Z and B (LNCS)*. Springer, 127–147.

J. Derrick and H. Wehrheim. 2005. Non-atomic Refinement in Z and CSP. In *ZB (LNCS)*, H. Treharne, S. King, M. C. Henson, and S. A. Schneider (Eds.), Vol. 3455. Springer, 24–44.

S. Doherty. 2003. *Modelling and verifying non-blocking algorithms that use dynamically allocated memory*. Master's thesis. Victoria University of Wellington.

S. Doherty, D. Detlefs, L. Groves, C. H. Flood, V. Luchangco, P. A. Martin, M. Moir, N. Shavit, and G. Steel. 2004a. DCAS is not a silver bullet for nonblocking algorithm design. In *SPAA*. 216–294.

S. Doherty, L. Groves, V. Luchangco, and M. Moir. 2004b. Formal Verification of a Practical Lock-Free Queue Algorithm. In *FORTE (LNCS)*, D. de Frutos-Escrig and M. Núñez (Eds.), Vol. 3235. Springer, 97–114.

B. Dongol. 2006. Formalising Progress Properties of Non-blocking Programs.. In *ICFEM (LNCS)*, Z. Liu and J. He (Eds.), Vol. 4260. Springer, 284–303.

B. Dongol. 2009. *Progress-based verification and derivation of concurrent programs*. Ph.D. Dissertation. The University of Queensland.

B. Dongol and J. Derrick. 2013. Simplifying proofs of linearisability using layers of abstraction. *ECEASST* 66 (2013).

B. Dongol, J. Derrick, L. Groves, and G. Smith. 2015. Defining Correctness Conditions for Concurrent Objects in Multicore Architectures. In *ECOOP (LIPIcs)*, J. T. Boyland (Ed.), Vol. 37. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 470–494. DOI:http://dx.doi.org/10.4230/LIPIcs.ECOOP.2015.470

C. Dragoi, A. Gupta, and T. A. Henzinger. 2013. Automatic Linearizability Proofs of Concurrent Objects with Cooperating Updates. In *CAV (LNCS)*, N. Sharygina and H. Veith (Eds.), Vol. 8044. Springer, 174–190.

R. Drexler, W. Reif, G. Schellhorn, K. Stenzel, W. Stephan, and A. Wolpers. 1993. The KIV System: A Tool for Formal Program Development. In *STACS (LNCS)*, P. Enjalbert, A. Finkel, and K. W. Wagner (Eds.), Vol. 665. 704–705.

T. Elmas. 2010. QED: a proof system based on reduction and abstraction for the static verification of concurrent software. In *ICSE (2)*, J. Kramer, J. Bishop, P. T. Devanbu, and S. Uchitel (Eds.). ACM, 507–508.

T. Elmas, S. Qadeer, A. Sezgin, O. Subasi, and S. Tasiran. 2010. Simplifying Linearizability Proofs with Reduction and Abstraction. In *TACAS (LNCS)*, J. Esparza and R. Majumdar (Eds.), Vol. 6015. Springer, 296–311.

T. Elmas, S. Qadeer, and S. Tasiran. 2009. A calculus of atomic actions. In *POPL*, Z. Shao and B. C. Pierce (Eds.). ACM, 2–15.

I. Filipović, P. W. O'Hearn, N. Rinetzky, and H. Yang. 2010. Abstraction for concurrent objects. *Theor. Comput. Sci.* 411, 51-52 (2010), 4379–4398.

K. Fraser and T. L. Harris. 2007. Concurrent programming without locks. *ACM Trans. Comput. Syst.* 25, 2 (2007).

D. Friggens. 2013. *On the Use of Model Checking for the Bounded and Unbounded Verification of Nonblocking Concurrent Data Structures*. Ph.D. Dissertation. Victoria University of Wellington. http://hdl.handle.net/10063/2702

H. Gao, Y. Fu, and W. H. Hesselink. 2009. Verification of a Lock-Free Implementation of Multiword LL/SC Object. In *DASC*. IEEE, 31–36.

H. Gao, J. F. Groote, and W. H. Hesselink. 2005. Lock-free dynamic hash tables with open addressing. *Distributed Computing* 18, 1 (2005), 21–42.

H. Gao, J. F. Groote, and W. H. Hesselink. 2007. Lock-free parallel and concurrent garbage collection by mark&sweep. *Sci. Comput. Program.* 64, 3 (2007), 341–374.

H. Gao and W. H. Hesselink. 2007. A general lock-free algorithm using compare-and-swap. *Inf. Comp.* 205, 2 (2007), 225–241.

P. H. B. Gardiner and C. Morgan. 1993. A Single Complete Rule for Data Refinement. *Formal Aspects of Computing* 5, 4 (1993), 367–382.

A. Gotsman, B. Cook, M. J. Parkinson, and V. Vafeiadis. 2009. Proving that non-blocking algorithms don't block. In *POPL*, Z. Shao and B. C. Pierce (Eds.). ACM, 16–28. DOI:http://dx.doi.org/10.1145/1480881.1480886

A. Gotsman and H. Yang. 2012. Linearizability with Ownership Transfer. In *CONCUR (LNCS)*, M. Koutny and I. Ulidowski (Eds.), Vol. 7454. Springer, 256–271.

A. Gotsman and H. Yang. 2013. Linearizability with Ownership Transfer. *Logical Methods in Computer Science* 9, 3 (2013).

L. Groves. 2007. Reasoning about Nonblocking Concurrency using Reduction. In *ICECCS*. IEEE Comp. Soc., 107–116.

L. Groves. 2008a. Trace-based Derivation of a Lock-Free Queue Algorithm. *Elect. Notes Theo. Comp. Sci.* 201 (2008), 69–98.

L. Groves. 2008b. Verifying Michael and Scott's Lock-Free Queue Algorithm using Trace Reduction. In *CATS (CRPIT)*, J. Harland and P. Manyem (Eds.), Vol. 77. 133–142.

L. Groves. 2009. Reasoning about Nonblocking Concurrency. *JUCS* 15, 1 (Jan 2009), 72–111.

L. Groves and R. Colvin. 2007. Derivation of a Scalable Lock-Free Stack Algorithm. *Electr. Notes Theor. Comput. Sci.* 187 (2007), 55–74.

L. Groves and R. Colvin. 2009. Trace-based derivation of a scalable lock-free stack algorithm. *Formal Asp. Comput.* 21, 1-2 (2009), 187–223.

R. Guerraoui and A. Schiper. 1996. Fault-tolerance by replication in distributed systems. In *Reliable Software Technologies Ada-Europe '96*, Al. Strohmeier (Ed.). LNCS, Vol. 1088. Springer, 38–57.

T. L. Harris. 2001. A Pragmatic Implementation of Non-blocking Linked-Lists. In *DISC (LNCS)*, J. L. Welch (Ed.), Vol. 2180. Springer, 300–314.

T. L. Harris, K. Fraser, and I. A. Pratt. 2002. A Practical Multi-word Compare-and-Swap Operation. In *DISC (LNCS)*, D. Malkhi (Ed.), Vol. 2508. Springer, 265–279.

I. J. Hayes, A. Burns, B. Dongol, and C. B. Jones. 2013. Comparing Degrees of Non-Determinism in Expression Evaluation. *Comput. J.* 56, 6 (2013), 741–755.

S. Heller, M. Herlihy, V. Luchangco, M. Moir, W. N. Scherer III, and N. Shavit. 2007. A Lazy Concurrent List-Based Set Algorithm. *Parallel Processing Letters* 17, 4 (2007), 411–424.

D. Hendler, N. Shavit, and L. Yerushalmi. 2010. A scalable lock-free stack algorithm. *J. Parallel Distrib. Comput.* 70, 1 (2010), 1–12.

T. A. Henzinger, C. M. Kirsch, H. Payer, A. Sezgin, and A. Sokolova. 2013a. Quantitative relaxation of concurrent data structures. In *POPL*, R. Giacobazzi and R. Cousot (Eds.). ACM, 317–328.

T. A. Henzinger, A. Sezgin, and V. Vafeiadis. 2013b. Aspect-Oriented Linearizability Proofs. In *CONCUR (LNCS)*, P. R. D'Argenio and H. C. Melgratti (Eds.), Vol. 8052. Springer, 242–256.

M. Herlihy and N. Shavit. 2008. *The art of multiprocessor programming*. Morgan Kaufmann. I–XX, 1–508 pages.

M. Herlihy and N. Shavit. 2011. On the Nature of Progress. In *OPODIS (LNCS)*, A. Fernández Anta, G. Lipari, and M. Roy (Eds.), Vol. 7109. Springer, 313–328.

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

R. Jagadeesan and J. Riely. 2014. Between Linearizability and Quiescent Consistency - Quantitative Quiescent Consistency. In *ICALP (LNCS)*, J. Esparza, P. Fraigniaud, T. Husfeldt, and E. Koutsoupias (Eds.), Vol. 8573. Springer, 220–231.

C. B. Jones. 1983. Tentative steps toward a development method for interfering programs. *ACM Trans. Prog. Lang. and Syst.* 5, 4 (1983), 596–619.

N. D. Jones and S. S. Muchnick. 1982. A Flexible Approach to Interprocedural Data Flow Analysis and Programs with Recursive Data Structures. In *POPL*, R. A. DeMillo (Ed.). ACM Press, 66–74.

B. Jonsson. 2012. Using refinement calculus techniques to prove linearizability. *Formal Asp. Comp.* 24, 4-6 (2012), 537–554.

L. Lamport. 1997. How to Make a Correct Multiprocess Program Execute Correctly on a Multiprocessor. *IEEE Trans. Computers* 46, 7 (1997), 779–782.

H. Liang and X. Feng. 2013a. *Modular verification of linearizability with non-fixed linearization points*. Technical Report. USTC. http://kyhcs.ustcsz.edu.cn/relconcur/lin

H. Liang and X. Feng. 2013b. Modular verification of linearizability with non-fixed linearization points. In *PLDI*, H.-J. Boehm and C. Flanagan (Eds.). ACM, 459–470.

H. Liang, X. Feng, and M. Fu. 2012. A rely-guarantee-based simulation for verifying concurrent program transformations. In *POPL*, J. Field and M. Hicks (Eds.). ACM, 455–468.

H. Liang, J. Hoffmann, X. Feng, and Z. Shao. 2013. Characterizing Progress Properties of Concurrent Objects via Contextual Refinements. In *CONCUR (LNCS)*, P. R. D'Argenio and H. C. Melgratti (Eds.), Vol. 8052. Springer, 227–241. DOI:http://dx.doi.org/10.1007/978-3-642-40184-8_17

R. J. Lipton. 1975. Reduction: a method of proving properties of parallel programs. *Commun. ACM* 18, 12 (1975), 717–721.

Y. Liu, W. Chen, Y. A. Liu, and J. Sun. 2009. Model Checking Linearizability via Refinement. In *FM (LNCS)*, A. Cavalcanti and D. Dams (Eds.), Vol. 5850. Springer, 321–337.

Y. Liu, W. Chen, Y. A. Liu, J. Sun, S. J. Zhang, and J. Song Dong. 2013. Verifying Linearizability via Optimized Refinement Checking. *IEEE Trans. Software Eng.* 39, 7 (2013), 1018–1039.

N. Lynch and M. Tuttle. 1989. An Introduction to Input/Output automata. *CWI-Quarterly* 2, 3 (1989), 219–246.

N. A. Lynch. 1996. *Distributed Algorithms*. Morgan Kaufmann.

N. A. Lynch and F. W. Vaandrager. 1995. Forward and Backward Simulations: I. Untimed Systems. *Inf. Comput.* 121, 2 (1995), 214–233. DOI:http://dx.doi.org/10.1006/inco.1995.1134

H. Massalin and C. Pu. 1992. A Lock-Free Multiprocessor OS Kernel (Abstract). *Operating Systems Review* 26, 2 (1992), 8.

M. Michael and M. L. Scott. 1996. Simple, Fast, and Practical Non-Blocking and Blocking Concurrent Queue Algorithms. In *PODC*. 267–275.

M. M. Michael. 2002. High performance dynamic lock-free hash tables and list-based sets. In *SPAA*. 73–82.

M. Moir, D. Nussbaum, O. Shalev, and N. Shavit. 2005a. Using elimination to implement scalable and lock-free FIFO queues. In *SPAA*, P. B. Gibbons and P. G. Spirakis (Eds.). ACM, 253–262.

M. Moir, D. Nussbaum, O. Shalev, and N. Shavit. 2005b. Using elimination to implement scalable and lock-free FIFO queues. In *SPAA*, P. B. Gibbons and P. G. Spirakis (Eds.). ACM, 253–262.

M. Moir and N. Shavit. 2007. Concurrent Data Structures. In *Handbook of Data Structures and Applications, D. Metha and S. Sahni Editors*. 47–14 47–30. Chapman and Hall/CRC Press.

C. Morgan and T. Vickers. 1992. *On the Refinement Calculus*. Springer-Verlag New York, Inc., Secaucus, NJ, USA.

B. C. Moszkowski. 1997. Compositional Reasoning Using Interval Temporal Logic and Tempura. In *COMPOS (LNCS)*, W. P. de Roever, H. Langmaack, and A. Pnueli (Eds.), Vol. 1536. Springer, 439–464.

B. C. Moszkowski. 2000. A Complete Axiomatization of Interval Temporal Logic with Infinite Time. In *LICS*. 241–252.

P. W. O'Hearn, J. C. Reynolds, and H. Yang. 2001. Local Reasoning about Programs that Alter Data Structures. In *CSL (LNCS)*, L. Fribourg (Ed.), Vol. 2142. Springer, 1–19.

P. W. O'Hearn, N. Rinetzky, M. T. Vechev, E. Yahav, and G. Yorsh. 2010a. Verifying linearizability with hindsight. In *PODC*, A. W. Richa and R. Guerraoui (Eds.). ACM, 85–94.

P. W. O'Hearn, N. Rinetzky, M. T. Vechev, E. Yahav, and G. Yorsh. 2010b. *Verifying linearizability with hindsight*. Technical Report. Queen Mary University of London.

R. Oshman and N. Shavit. 2013. The SkipTrie: low-depth concurrent search without rebalancing. In *PODC*, P. Fatourou and G. Taubenfeld (Eds.). ACM, 23–32.

S. S. Owicki and D. Gries. 1976. An Axiomatic Proof Technique for Parallel Programs I. *Acta Inf.* 6 (1976), 319–340.

S. Owre, S. Rajan, J. M. Rushby, N. Shankar, and M. K. Srivas. 1996. PVS: Combining specification, proof checking, and model checking. In *CAV*, R. Alur and T. A. Henzinger (Eds.), Vol. 1102. Springer, 411–414.

F. Pacull and A. Sandoz. 1993. R-linearizability: an extension of linearizability to replicated objects. In *Fourth Workshop on Future Trends of Distributed Computing Systems*. 347–353.

K. Ramamritham and P. K. Chrysanthis. 1992. In Search of Acceptability Citeria: Database Consistency Requirements and Transaction Correctness properties. In *IWDOM*, M. T. Özsu, U. Dayal, and P. Valduriez (Eds.). Morgan Kaufmann, 212–230.

M. Raynal. 2013. *Distributed Algorithms for Message-Passing Systems*. Springer, Chapter Atomic Consistency (Linearizability), 427–446.

J. C. Reynolds. 2002. Separation Logic: A Logic for Shared Mutable Data Structures. In *LICS*. IEEE Computer Society, 55–74.

G. Schellhorn, J. Derrick, and H. Wehrheim. 2014. A Sound and Complete Proof Technique for Linearizability of Concurrent Data Structures. *ACM Trans. Comput. Logic* 15, 4, Article 31 (Sept. 2014), 37 pages.

G. Schellhorn, B. Tofan, G. Ernst, and W. Reif. 2011. Interleaved Programs and Rely-Guarantee Reasoning with ITL. In *TIME*, C. Combi, M. Leucker, and F. Wolter (Eds.). IEEE, 99–106.

G. Schellhorn, H. Wehrheim, and J. Derrick. 2012. How to Prove Algorithms Linearisable. In *CAV (LNCS)*, P. Madhusudan and S. A. Seshia (Eds.), Vol. 7358. Springer, 243–259.

C.-H. Shann, T.-L. Huang, and C. Chen. 2000. A Practical Nonblocking Queue Algorithm Using Compare-and-Swap. In *ICPADS*. 470–475.

M. Shapiro and B. Kemme. 2009. Eventual Consistency. In *Encyclopedia of Database Systems*, L. Liu and M. T. Özsu (Eds.). Springer US, 1071–1072.

N. Shavit. 2011. Data structures in the multicore age. *Commun. ACM* 54, 3 (2011), 76–84.

G. Smith. 1999. *The Object-Z Specification Language*. Springer US.

G. Smith, J. Derrick, and B. Dongol. 2014. Admit Your Weakness: Verifying Correctness on TSO Architectures. In *FACS (LNCS)*, I. Lanese and E. Madelaine (Eds.), Vol. 8997. Springer, 364–383.

B. Tofan, S. Bäumler, G. Schellhorn, and W. Reif. 2010. Temporal Logic Verification of Lock-Freedom. In *MPC (LNCS)*, C. Bolduc, J. Desharnais, and B. Ktari (Eds.), Vol. 6120. Springer, 377–396.

B. Tofan, G. Schellhorn, and W. Reif. 2011. Formal Verification of a Lock-Free Stack with Hazard Pointers. In *ICTAC (LNCS)*, A. Cerone and P. Pihlajasaari (Eds.), Vol. 6916. Springer, 239–255.

B. Tofan, G. Schellhorn, and W. Reif. 2014. A Compositional Proof Method for Linearizability Applied to a Wait-Free Multiset. In *iFM (LNCS)*, E. Albert and E. Sekerinski (Eds.), Vol. 8739. Springer, 357–372.

R. K. Treiber. 1986. *Systems programming: Coping with parallelism*. Technical Report RJ 5118. IBM Almaden Res. Ctr.

A. J. Turon and M. Wand. 2011. A separation logic for refining concurrent objects. In *POPL*, T. Ball and M. Sagiv (Eds.). ACM, 247–258.

V. Vafeiadis. 2007. *Modular fine-grained concurrency verification*. Ph.D. Dissertation. University of Cambridge.

V. Vafeiadis. 2009. Shape-Value Abstraction for Verifying Linearizability. In *VMCAI (LNCS)*, N. D. Jones and M. Müller-Olm (Eds.), Vol. 5403. Springer, 335–348.

V. Vafeiadis. 2010. Automatically Proving Linearizability. In *CAV (LNCS)*, T. Touili, B. Cook, and P. Jackson (Eds.), Vol. 6174. Springer, 450–464.

V. Vafeiadis, M. Herlihy, T. Hoare, and M. Shapiro. 2006. Proving correctness of highly-concurrent linearisable objects. In *PPOPP*, J. Torrellas and S. Chatterjee (Eds.). 129–136.

V. Vafeiadis and M. J. Parkinson. 2007. A Marriage of Rely/Guarantee and Separation Logic. In *CONCUR (LNCS)*, L. Caires and V. T. Vasconcelos (Eds.), Vol. 4703. Springer, 256–271.

S. van Staden. 2015. On Rely-Guarantee Reasoning. In *MPC (LNCS)*, R. Hinze and J. Voigtländer (Eds.), Vol. 9129. Springer, 30–49. DOI:http://dx.doi.org/10.1007/978-3-319-19797-5_2

M. T. Vechev and E. Yahav. 2008. Deriving linearizable fine-grained concurrent objects. In *PLDI*, R. Gupta and S. P. Amarasinghe (Eds.). ACM, 125–135.

M. T. Vechev, E. Yahav, and G. Yorsh. 2009. Experience with Model Checking Linearizability. In *SPIN (LNCS)*, C. S. Pasareanu (Ed.), Vol. 5578. Springer, 261–278.

Z. Zhang, X. Feng, M. Fu, Z. Shao, and Y. Li. 2012. A Structural Approach to Prophecy Variables. In *TAMC (LNCS)*, M. Agrawal, S. B. Cooper, and A. Li (Eds.), Vol. 7287. Springer, 61–71.